Analysing Dynamic Memory Allocation in Embedded Systems for Multi-Channel Modems

Matthias Peintner, Faheem Bukhatwa and Ahmed Patel
Computer Networks & Distributed Systems Research Group
Department of Computer Science
University College Dublin
Belfield, Dublin 4, Ireland
Tel. +353-1-7162476; Fax +353-1-2697262
Corresponding author email : *faheemfb@gmail.com*

Abstract

High performance data communication equipment is essential to support the increasing number of users and traffic on the Internet. One important data communication component is an embedded system running multi-channel data modem software. Such an embedded system is used in IP network concentrators. In this paper, we compare four dynamic memory allocation algorithms to increase performance of the system. We examine different implementation mechanisms for First fit and Best fit. We implement them and compare their functionality using a simulation model of the real application running on the embedded system.

1. Introduction

Although technologies such as Asymmetric Digital Subscriber Line (ADSL) and cable modems are increasingly common for Internet access, very large numbers of users still depend on dial-up modem connections over telephone lines, and this will continue to be the case for the foreseeable future due to the low cost and ready availability of such connections. The design of equipment to support such users is therefore of continuing interest and importance.

The equipment must be able to support large numbers of users with varying requirements for connection protocols determined by their available hardware and software. This leads to a need for modem concentrator systems which can deal efficiently with many simultaneous connections. A critical resource in the operation of such a system is the available memory, which must be allocated and deallocated dynamically as the users connect and disconnect. In this paper dynamic memory allocation in an embedded data modem software system is examined. We begin by presenting an overview of the multi-channel data modem software. The dynamic memory allocator algorithms used in this work is then described. Then a description is outlined of the implementation mechanisms that were used to optimise the allocators to maximise the effective used memory and to minimise response time of the dynamic memory allocation algorithms. Lastly performance results of the dynamic memory allocators with a software model are presented.

2. Multi-Channel Data Modem Software

The performance of modern digital signal processing (DSP) chips is good enough to allow them to execute multiple instances of the modem software at the same time. We refer to each such instance of the software as a modem instance. Each modem instance corresponds to one hardware port for data exchange with the users. The data received from the user is passed to a router for onward transmission on higher speed connections.

The Multi-channel data modem software (MCDMS) deals with five data modem instances in parallel. Each data modem instance starts from an idle state in which it is waiting for a dial up connection. When a user calls the modem instance, the user's

software and the modem software agree protocols for data transfer. These session protocols are valid for this session only. The user may choose from a set of protocols provided by the modem software. Establishing a session (i.e. a connection) follows a strict order of predefined steps. There are restrictions on the allowed combinations of protocols: for example, the HDLC protocol cannot be established if the V.110 protocol is established, and the MNP5 protocol can only be established if one of V.32bis, V.34 or V.90 is already established. This leads to the software state machine illustrated in Fig. 1.

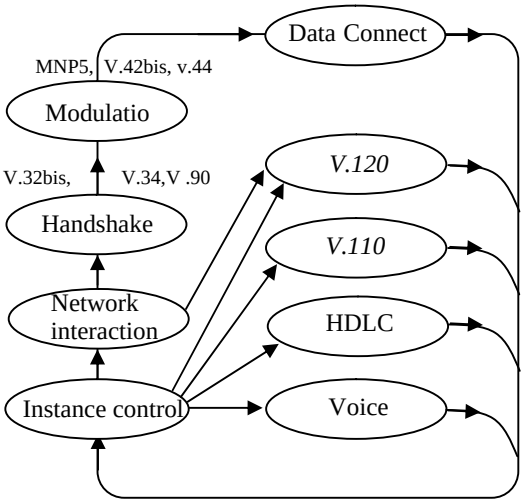

Fig. 1: Single instance modem software units

The progress of a typical call is as follows. Initially, the modem instance is in an idle state, termed Instance control. On receipt of a call the modem instance changes state to one of the connection established states according to the session protocol(s) requested by the user. In this state the modem instance and the dial-up user have acknowledged a connection and all necessary session protocols are established. The modem instance is now dealing with the connection, providing data transfer between the router and the dial-up user. This continues until one of the participants, typically the user, cancels the connection. The modem instance then returns to the idle state. The modem software is structured into units, each of which corresponds to one of the states in Fig. 1.

3. Dynamic Memory Management

In this paper we focus on a dynamic memory management system for an embedded system. This real time system is running multi channel data communication software. Strict and urgent performance requirements are placed on software used on embedded systems. Such systems are process time dependent, they have to meet real time constraints of the execution process, because they are applied to take control over time critical real-world environments.

The allocation policy has to be highly efficient in terms of memory usage to meet resource constraints. This is done by keeping the actual fragmentation of the memory low. Fragmented blocks, are gaps in memory that are smaller than the actual memory size requested by a program.

Another important term is to keep the memory space overhead of the allocator low. This space overhead is required to keep track of block sizes and whether blocks are allocated or free etc. The implemented software has to meet the strict and urgent real-time system requirements. The allocation software has to return to a stable state at any stage without breaking a certain time limit.

The embedded system has to be reliable. The algorithm has to block states, which might lead to an error later on. For example, if nearly all memory resources are used the algorithm will block low priority jobs in order to increase resources for high priority jobs. Such scenarios are known as fault-tolerant constraints. Ideally the embedded system application should be set up once and then run without error until the world stops turning or the hardware is put out of service.

4. The Algorithms

A large number of dynamic memory allocation algorithms are available, and it is impractical to evaluate all of them. In this work we focus on the most promising allocation policies, namely: **first fit** and **best fit**.

These two allocation policies were chosen to investigate through state of the art technology in

dynamic memory allocation [2,6]. This survey proposed different implementation mechanisms that optimise the allocators to different criteria and compromises of those. Criteria for dynamic memory fragmentation are: allocation speed, freeing speed, memory fragmentation and effective used memory.

In First fit allocation policy, when a request for memory is received, the First Fit algorithm simply searches the list of free blocks of memory from the beginning. The first found data block large enough to satisfy the request is used. The block is marked as "used". If the block is larger than needed, it is split and the remainder is put back on the free list. If no data block large enough is found in the list, then the request is refused. When a block is released the allocator marks the block as being "unused" and the block is placed back on the free list.

In Best Fit policy, when a request for memory is received, the Best Fit allocator searches the ordered list of free blocks for the block that minimises the remainder. The allocator then marks that block as "used". If the block is larger than needed, the remainder is put back on the free list. If no block large enough is found in the list, then the request is refused. When a block is released the allocator marks the block as being "unused" and the block is placed back on the free list.

4.3. Choice of an implementation mechanism

Choosing a good implementation strategy to match the memory allocation strategy of the application program is one part of a good allocator. Choosing the right mechanisms for implementing the strategies is as important for an allocator to match the crucial criteria of the application program [6]. With well-behaved implementation mechanisms the allocator can be optimised to maximise the effective used memory, keep fragmentation low and keep response time below an acceptable limit. This is the ranking of the crucial criteria on our embedded system. The following implementation mechanisms are used to fulfill the named problematic nature.

4.3.1. Header fields. In this work header fields are chosen to keep track of the used and unused chunks of memory [5,6]. Header fields have been used for many implementations of allocators. They are proved to work well and efficiently. They are relatively easy to use and handle. The major disadvantage in comparison to bitmaps is that headers have a larger space overhead [6].

The header field may be as large as one machine word. Any smaller field makes no sense, since one machine word is the smallest amount of memory available. Any larger field should be avoided because it would increase the implementation overhead dramatically. This header field should be sufficient to store the size of the block, a flag to indicate whether the block is used or unused. More unused bits may be used to hold other helpful information in the form of flags.

On our embedded system a machine word is 32 bits. This is large enough to record the size of a block plus some additional flags. For example, when using two one-bit flags, the size field in the header field is still large enough to record a maximum block size of 1 Gbytes. The maximum block size must be equal to the heap memory size, since at the start, the whole heap is only one large unused block. The expected memory sizes are about 128 Kbytes. Therefore, only 17 bits are needed to record the block size. This gives high flexibility when using a 32-bit word. 15 bits are available for one-bit flags, additional information.

Such header fields for each block have an acceptable low space overhead. This keeps the memory effectively usable to store user data fairly high. This was indicated as the most crucial optimisation criteria.

4.3.2. Address ordered free list. This header field already implements an address ordered linked list, needed by the First fit and Best fit allocator to search for the first or Best fitting block. Recording the size of each block in the header field makes it easy to calculate the start of the following block, i.e. to create a linked list. Every block has a header indicating the size of the block. The pointer to the block plus the block size gives the pointer to the start of the following block. No further data structure is needed to manage the blocks in memory. Each block can be reached from the start block in memory. A search in this address ordered list steps through all used and unused blocks.

The search stops as soon as a first or Best fitting free block is found. A pointer to the start address of a block is calculated as: $block_{i+1} = block_i + size(block_i)$, fig. 3 illustrates this example.

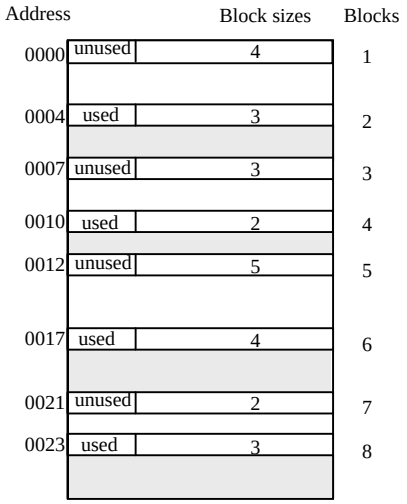

Fig. 3: Size field creates a linked list

An address ordering of free blocks is the most efficient strategy for *First fit* and *Best fit* to minimise memory fragmentation [2]. Keeping memory fragmentation low was named as second most important criteria for our dynamic memory allocator. Using the linear list linked up by the header fields has a major disadvantage. Searching for a free block also visits used blocks, although these blocks are not relevant to the search. This problem could be solved by using link fields within blocks [6].

4.3.3. Coalescing. Coalescing is the way in which adjacent free blocks in memory are merged into a single free block. Immediate Coalescing tries to merge blocks whenever a block is freed. This sub-strategy has an additional calculation overhead, since blocks have to be split more often if blocks of a specific size are requested and freed more frequently.

In the sub-strategy Deferred Coalescing, a freed block is simply marked as "unused" with out merging. If a block of a specific size is requested, the allocator searches the address ordered free list. If a large enough block is not found, the list is searched a second time. This time the allocator tries to merge adjacent free blocks, in order to get a large enough block to satisfy the request. Deferred Coalescing saves processing time if some sizes are very commonly used, but it may have an affect on memory fragmentation. Johnstone [2] investigated the deferred coalescing and found that it has a bad impact on memory fragmentation for most common application programs. Therefore we will use only the Immediate Coalescing strategy

4.3.4. Boundary tags. Coalescing performs merging of free blocks in memory. Every time a block is freed its previous and following blocks in memory are visited. Adjacent free blocks are merged into one free block. Coalescing has to examine the previous and following blocks of the currently freed block in memory. Therefore the start addresses of these blocks are needed.

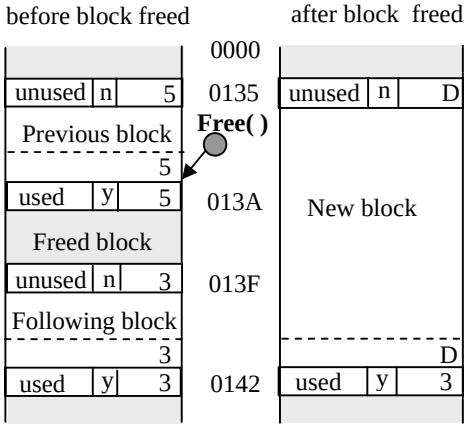

Fig. 4: Boundary and entry of a free block.

The start address of the next block is calculated by adding the start address of the freed block and its size. To get the address of the previous block, the linear list must be searched from the beginning. This search can take a long time. Boundary tags are a good mechanism to eliminate this search time. Each header field gets a one-bit flag indicates whether the previous block is used or not, called: previous used flag. If the previous block is free its size can be stored in the last word of the block, see fig. 4. The start address of the previous free block is:

$$Pointer_{\text{prev block}} = pointer_{\text{free block}} - size_{\text{prev block}}$$

The speeding up process of the free function is economic. There is no additional space overhead required. Only a computation overhead of a function that writes the block size into the end of the block. The previous used flag and the boundary entry together are called boundary tag. This boundary tag is the key that speeds up freeing of a block to a constant time rather than a search time that depends on the number of entries. We will refer to this as Fast Free. When combined with First fit or Best fit algorithms, we shall refer to them as First fit fast free and Best fit fast free consecutively. Freeing the blocks without the use of boundary tags will be referred to as the Slow Free.

Fig. 4 illustrates the freeing of a block. First the block at address 013A is tagged as free. Because the previous block used flag indicates it is free (y). The previous block and the just freed block can be merged as a single free block of size 10 starting from address 0135. The start address of the previous block is calculated as explained: 013A - 5 = 0135. The next step examines the following block. Since it is not being used, it is merged with the other free blocks. The header field and the boundary tag of the merged free block must be updated.

4.4. Implemented Algorithms

The Choice of both the First fit and Best fit strategies with the Boundary tag implementation mechanisms, led to the implementation of 4 different dynamic memory allocation algorithms, namely:

- First fit address ordered free list (First fit)
- First fit address ordered free list with boundary tags (First fit fast free)
- Best fit address ordered free list (Best fit)
- Best fit address ordered free list (Best fit fast free).

These four dynamic memory allocators were tested with a model of the software running on the embedded system emulating the memory request/reclamation stream of the real program. The allocators where evaluated according to the following 3 criteria, the most important first: (a) minimise memory fragmentation. (b) maximise the effective used memory and (c) keep the response time of the dynamic memory allocator low.

5. Evaluating Memory Allocators

In this section the four allocators are evaluated with a model of the software running on the embedded system. This model simulates software units of the real application in the way they request and release memory. The model creates a data structure with memory request and memory release instructions for each software unit and links this data structures in a manner similar to the way software units are connected in a real application.

A long term run of the real application software is simulated using this model and calculations are made of memory fragmentation, memory used to effectively store data for the user program and response time of the allocator at crucial states. Crucial states for memory fragmentation and effective used memory are: when the amount of data allocated in the memory is at a maximum [2]. Crucial states for response time are: when the number of blocks in the memory is at a maximum.

Table 1: Performance of main memory use measured with the 4 allocators

Implementation of allocator algorithm	First fit		Best fit	
	slow free	slow free	slow free	slow free
Physical size (bytes)	30420	30420	30464	30464
Effective used (bytes)	29788	29796	29788	29788
Headers (bytes)	532	532	532	532
Fragments (bytes)	100	92	144	144
Fragments of memory %	0.33%	0.30%	0.47%	0.47%

5.1 Efficiency of main memory use

Table 1 gives the simulation results of main memory use for the four different memory allocators. The row “Physical size” shows the size of the simulated physical main memory. This is the amount of main memory that was available to deal with all the memory requests of the modelled program. The row “Effective used” gives the size of all allocated blocks in memory. “Headers” gives the size of all header

fields in memory. “Fragmented” gives the size of all holes in memory. All these parameters were recorded at the point of maximum memory use.

It was not surprising that the two First fit allocators were able to handle all memory requests in memory pools of similar size, because both allocators used the same strategy. For the two Best fit allocators similar results were observed because both used the same allocation strategy and have the same physical memory requirement. The discrepancy between effective used memory and fragmented memory of the First fit allocators occurred because First fit slow free has more internal fragmentation, whereas First fit fast free has more external fragmentation [4].

Fig. 5 gives an easily comparable illustration of the performance parameters in relative values. As the figure shows, all four implementations are very efficient and very similar. All allocators have less than 0.5% actual fragmentation. Only 1.8% of memory is needed by the worst allocator for managing the memory resource. All allocators use more than 97.8% of memory to store actual user program data.

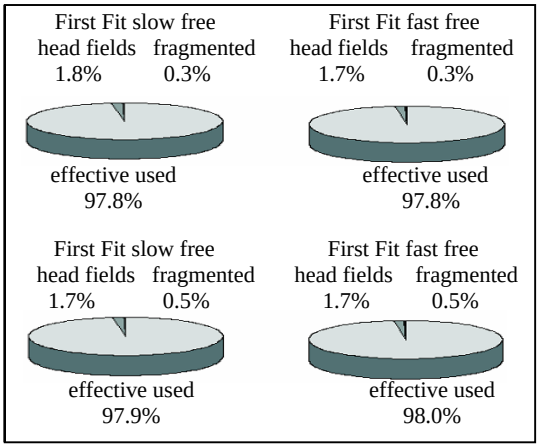

Fig. 5: Efficiency of main memory use

The following result were obtained for the worst behaving allocators, that is the Best fit: (a) efficiency of memory use: 97.8%, (b) memory used for header fields: 1.7% and (c) memory fragmented: 0.5%. These are considered to be good results and comparable to results measured by others; see the Discussion below.

Table 2 gives the average response time of the memory allocation function malloc( ) and of the memory reclamation function free( ). The average time used to manage a block is the sum of the times spent allocating and reclaiming a block. In the table it is named as avg. total time. Finally the response time of the swapping function swap( ) is given. All times given are in microseconds

Table 2: Average response time in the memory management functions

Implementation of allocator algorithm	First fit		Best fit	
	slow free	fast free	slow free	fast free
Avg. mem. allocation (μs)	2.80	2.89	2.72	3.24
Avg. freeing Time (μs)	1.28	0.24	1.42	0.36
Avg. total Time (μs)	4.08	3.13	4.14	3.60
Avg. swap Time (μs)	128.88	134.33	138.36	144.28

5.2. Time Efficiency of allocation functions

The First fit strategy takes less time to manage memory than the Best fit strategy. In absolute terms, the slowest strategy Best fit slow free takes 4.14 μs to allocate a block in memory and reclaim it. The best strategy, First fit fast free, is indisputably faster. The slow free strategy is 6.0 μs faster when performing swapping.

5.3 Size of the allocator implementations

Finally, the size of the executable allocator module is also of interest. The size of the compiled software is shown in Table 3. The “Text” row gives the size of the program text segment, “Data” row indicates the size of the initialised data segment and the “Bss” row gives the size of the uninitialised data segment. The “Sum” row gives the sum of all three program segments.

The largest module is less than 3.3 Kbytes. This is an acceptable size for a dynamic memory management system. The implemented versions with fast free need more memory than the basic allocator versions. The fast free implementations are about 470 bytes larger than the slow free implementations. This

is a noticeable but not significant difference. In practice the time saved during freeing outweighs the 470 additional bytes required for the executable module.

6. Discussion

The results presented above from the simulation show good performance for all four dynamic memory allocators. All of the allocators use more than 97.8% efficiency to store actual user data. In the worst case 1.7% is used for header fields and 0.5% memory fragmentation occurs. In absolute terms this is a fairly good performance. These performance parameters show no significant difference between the four implemented allocator strategies.

Measurements of response time give more insight in evaluating the best behaving allocator. First fit fast free is significantly faster in responding to a memory allocation or reclamation request. This allocator needs 3.13 µs to deal with a request for allocating and reallocating memory (2.89 µs to deal with an allocation request and 0.24 µs to deal with a reclamation request). The second best was Best fit fast free with a response time of 3.60 µs for a memory allocation and reclamation.

Table 3: Size of executable allocator implementation

Implementation of allocator algorithm	First fit		Best fit	
	slow free	fast free	slow free	fast free
Text (bytes)	2721	3193	2753	3225
Data (bytes)	72	72	72	72
Bss (bytes)	0	0	0	0
Sum (bytes)	2793	3265	2825	3297

In general, First fit is faster than Best fit and fast free is significantly faster than slow free. The measured differences depend on the actual workload. In absolute terms this is an acceptable response time for the fast free allocators, especially First fit fast free.

Comparing the size of executable allocator modules shows an acceptable size of 3.3Kbytes for First fit fast free and Best fit fast free. The module sizes of the slow free implementations are slightly smaller (2.8Kbytes) but the advantage in response time of the fast free allocators more than makes up for the larger space overhead of the executable.

The results show that First fit fast free is the most suitable allocator. There is no significant difference in the Efficiency of memory use. However, the First fit strategy benefits from less time spent in loops to find a suitable block. Furthermore, the subsidiary strategy fast free allows freeing to be done in constant time leading to a useful speed improvement on average.

Johnstone and Wilson [2] measured the performance of a wide variety of dynamic memory allocators with eight different user programs. In their framework, First fit and Best fit performed significantly better than other allocators. They also measured memory fragmentation for sixteen allocators with eight different application programs. On average they measured a memory fragmentation of 0.98% with a Best fit algorithm and 0.91% with a First fit algorithm in the best implementations. All implementation overheads of the chosen algorithms were eliminated in their study.

The results observed in our simulations show good agreement with the results measured by Johnstone and Wilson, despite the different environments considered. The First fit allocators showed memory fragmentation of 0.3% and the Best fit allocators showed memory fragmentation of 0.5%.

The next step in this work is to evaluate the performance of the allocators in the embedded system itself. It needs to be proofed whether real environment shows significant performance degradation in comparison to the simulated environment. An evaluation of the scalability of the allocators is also of interest. This could be addressed by further simulation work followed by experiments in the embedded environment. Further speed optimisation of the allocator algorithms could be done by using link fields to build up hidden search trees [6]. Adding fault-tolerance constraint to a well performing dynamic memory allocator would add great performance to the embedded system, this is untouched in this work yet.

7. Conclusion

In this paper we examined four different dynamic memory allocation algorithms for use in an embedded

system running multi-channel data modem software. We examined two different implementation mechanisms for two well-known dynamic memory allocation strategies, namely First fit and Best fit. The purpose of this investigation was to find the most suitable technique that can be used in an embedded system with rigid real-time operational constraints.

The comparison of the dynamic memory allocator algorithms was based on a model that simulates the dynamic storage use of the real application program on the embedded system. The allocation strategy First fit fast free was found to be the most suitable for use in the embedded system. It shows little memory fragmentation and has a rapid response time.

For further work, we recommend comparison of the results from the model to results measured with the real modem software.

8. References

[1] H. J. Bartsch. *Taschenbuch mathematischer Formeln, volume 18*. Fachbuchverlage Leipzig, 1998.

[2] Mark S. Johnstone and Paul R. Wilson. "*The memory fragmentation problem: solved?*" ACM SIGPLAN Notices, v.34 n.3 p.26-36, March 1999.

[3] Donald E. Knuth. *The Art of Computer Programming 1: Fundamental Algorithms*. Addison Wesley, 3rd edition, 1997.

[4] Matthias Peintner. "*Analysing dynamic memory management for a DSP*". M.Sc. thesis, Department of Computer Science, University College Dublin, October 2001.

[5] Thomas Standish. *Data Structure Techniques*. Addison-Wesley, Massachusetts, 1980.

[6] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. *Dynamic storage allocation: A survey and critical review*. In H. G. Baker (ed.), Proc. 1995 International Workshop on Memory Management, Kinross, UK, Springer Verlag, 1995.