Journal of Information Processing Vol.22 No.4 1–3 (July 2014)

[DOI: 10.2197/ipsjjip.22.1]

Regular Paper

Component-based Software Development Framework for Embedded IoT Devices（仮）

Takuro Yamamoto[1] Takuma Hara[2] Takuya Ishikawa[2] Hiroshi Oyama[3] Hiroaki Takada[2]
Takuya Azumi[1]

Received: March 4, 2014, Accepted: August 1, 2014

Abstract:

Keywords:

1. Introduction

2. System Model

3. Proposed Framework

3.1 TLSF+TECS

3.1.1 TLSF

TLSF (Two-Level Segregate Fit) memory allocator [1] [2] is a dynamic memory allocator suitable for the real-time system proposed by M. Masmano et al. TLSF memory allocator has the following features.

Real-time property

The worst execution time required for allocating and deallocating memory does not depend on the data size. TLSF always works with $O(1)$, and it is possible to estimate the response time.

Fast speed

In addition to being able to always estimate the worst execution time, TLSF is executed at high speed.

Efficient memory consumption

Memory efficiency is improved by suppressing memory fragmentation. Various tests have obtained an average fragmentation of less than 15% and a maximum fragmentation of less than 25%.

3.1.2 TLSF Algorithms

TLSF algorithm classifies memory blocks into two stages and searches for a memory block that is optimal for the requested memory size. The overview of TLSF algorithm is shown in Fig. 1. Consider a case where a request, *malloc*(100), is called to allocate a memory. In the first step, it is classified by the most significant bit of the requested memory size. In this case, since 100 is represented by binary number as 1100100, it is in the range from 64 to 128 from the most significant bit. In the second step, it is further classified. In this case, 64 to 128 are divided into 4, and 100 is in the block of 96 to 111. Free block [*1] in this range is used.

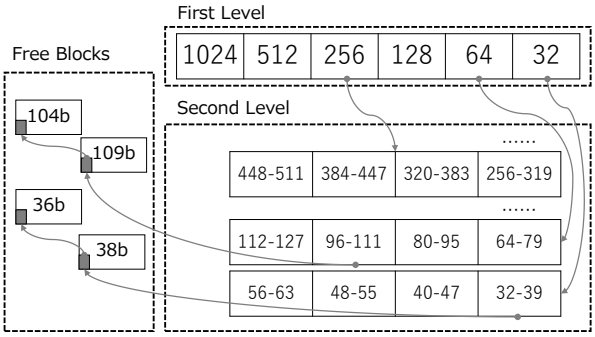

Fig. 1 TLSF Algorithm

```
1  signature sMalloc {
2      int initializeMemoryPool(void);
3      void *calloc( [in]size_t nelem,
4                    [in]size_t elem_size );
5      void *malloc( [in]size_t size );
6      void *realloc( [in]const void *ptr,
7                     [in]size_t new_size );
8      void free( [in]const void *ptr );
9  };
```

Fig. 2 Signature description of memory management

A simple fixed-size memory block allocator results in waste of up to 50%, but TLSF classifies it finely in two steps, so it is a memory efficient algorithm. In addition, TLSF searches at the same speed and at high speed, $O(1)$.

3.1.3 Component Design of TLSF memory allocator

This section describes the component design of TLSF memory allocator. In this research, we are using TECS to componentize TLSF. The version of TLSF used is 2.4.6[*2].

Fig. 2 is a signature description for memory management used by the allocator. It defines the memory pool initialization function *initializeMemoryPool*, memory allocation function *calloc*, *mal-*

1 Graduate School of Engineering Science, Osaka University
2 Graduate School of Information Science, Nagoya University
3 OKUMA Corporation

*1 Free block is an available memory block.
*2 http://www.gii.upv.es/tlsf/main/repo

```
1  celltype tTLSFMalloc {
2      [inline]
3          entry sMalloc eMalloc;
4      attr {
5          /* memory pool size in bytes */
6          size_t memoryPoolSize;
7      };
8      var {
9          [size_is( memoryPoolSize / 8 )]
10             uint64_t *pool;
11     };
12 };
```

Fig. 3 Celltype description of TLSF memory allocator component

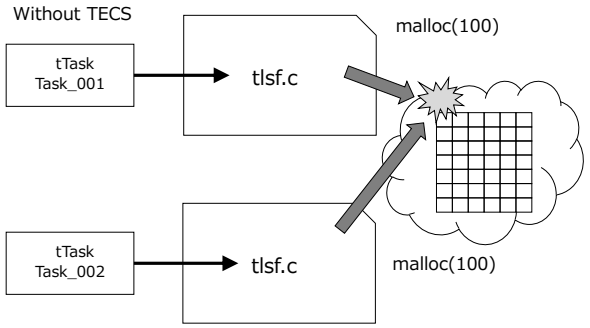

Fig. 4 TLSF before componentization

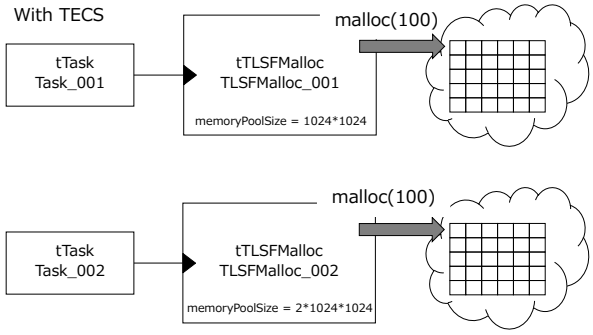

Fig. 5 TLSF after componentization

loc, *realloc*, and memory release function *free* as a signatures.

The celltype description of the TLSF memory allocator component is shown in Fig. 3. An entry port, *eMalloc*, is connected to all components that perform memory management such as *malloc* and *free*. Here, *[inline]* is a specifier for Implementation as inline functions. A memory pool size is defined as an attribute, and a pointer to a memory pool is defined as an variable. Each component holds its own heap area, so even when calling functions for memory management at the same time in different threads, it is possible to operate without memory contention.

As shown in Fig. 4, since TLSF before componentization shares the heap area with multiple threads, if memory is allocated or released simultaneously from multiple threads, memory competition may occur in some cases. In the research, TLSF is componentized using TECS as shown in Fig. 5. It is possible to operate in thread safe without exclusive control, because each component independently holds a heap area and manages memory within it.

Fig. 6 shows the build description of the TLSF memory allocator component shown in Fig. 5[*3]. Two sets of task components and TLSF components are combined. Each memory pool size can be configured as an variable (Lines 5 and 11 in Fig. 6). Fig. 7 is the part of the code actually calling the function of the TLSF memory allocator component. The use part shows a function that the mruby VM allocates memory in the mruby on TECS framework [3] [4] which is introduced in Section **??**. Lines 8 calls the *free* function of the TLSF memory allocator component. *cMalloc_* represents the name of the calle port (Lines 2 in Fig. 6). Likewise, Lines 13 and 17 call the function for memory allocation. The heap area of *TLSFMalloc_001* component is used if the code of 7 is executed in *Task_001*, and if that is executed in *Task_002*, the heap area of *TLSFMalloc_002* component is used, respectively. In this way, in the component-based development using TECS, it is possible to operate with the same code without modifying the C code, although the cells are different.

```
1  cell tTask Task_001 {
2      cMalloc = TLSFMalloc_001.eMalloc;
3  };
4  cell tTLSFMalloc TLSFMalloc_001 {
5      memoryPoolSize = 1024*1024; /* 1MB */
6  };
7  cell tTask Task_002 {
8      cMalloc = TLSFMalloc_002.eMalloc;
9  };
10 cell tTLSFMalloc TLSFMalloc_002 {
11     memoryPoolSize = 2*1024*1024; /* 2MB */
12 };
```

Fig. 6 Build description of TLSF memory allocator component

```
1  void*
2  mrb_TECS_allocf(mrb_state *mrb, void *p,
3                  size_t size, void *ud)
4  {
5      CELLCB *p_cellcb = (CELLCB *)ud;
6      if (size == 0) {
7          //tlsf_free(p);
8          cMalloc_free(p);
9          return NULL;
10     }
11     else if (p) {
12         //return tlsf_realloc(p, size);
13         return cMalloc_realloc(p, size);
14     }
15     else {
16         //return tlsf_malloc(size);
17         return cMalloc_malloc(size);
18     }
19 }
```

Fig. 7 Example of TLSF memory allocator component

4. Evaluation

5. Related Work

6. Conclusion

Acknowledgments

References

[1] Masmano, M., Ripoll, I., Crespo, A. and Real, J.: TLSF: a New Dynamic Memory Allocator for Real-Time Systems, *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 79–88

[*3] Other call/entry ports, attributes, and valuables are actually described, but it is omitted here for the simplicity.

(2004).
[2] : TLSF, http://www.gii.upv.es/tlsf/.
[3] Azumi, T., Nagahara, Y., Oyama, H. and Nishio, N.: mruby on TECS: Component-Based Framework for Running Script Program, *Proceedings of the 18th IEEE International Symposium on Real-Time Computing (ISORC)*, pp. 252–259 (2015).
[4] Yamamoto, T., Oyama, H. and Azumi, T.: Lightweight Ruby Framework for Improving Embedded Software Efficiency, *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 71–76 (2016).