Memory Management

A J-Component Project Report

Submitted by

Aditya Mishra (19BCE0342)
Mayank Raj Anand (19BCE0601)

Under the guidance of

SUDHA.S

in partial fulfillment for the award of the degree of

B. Tech.

in

COMPUTER SCIENCE AND ENGINEERING

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

1. Abstract

In this present of computing applications, there's a key got to manage the memory of computers. during this world one cannot live while not economical memory management system. currently one question comes in everyone's mind that what's memory management? Memory management is that the perform of package that manages and handles primary memory and moves processes at the same time between main memory and disk throughout execution, it keeps track of each memory location and is handled by heart Management Unit (MMU). can|it'll} decide that method will get memory at what time that is decision Memory Allocation. There square measure numerous styles of memory management system like contagious and non-contagious memory management systems. Here we tend to are attempting to check differing types of memory allocation algorithms that square measure employed in memory management for OS and RTOS.
There also are some current problems concerning memory management systems like premature frees and hanging pointers, memory leak, external fragmentation etc. currently we've more experienced many survey concerning memory management, and that we have additionally found some analysis papers and analyze them totally.
We have written their abstract, known downside associated created an overall conclusion.
By surfing these several analysis papers, we've come upon many alternative styles of memory management systems. then we've created associate analysis concerning differing types of memory Allocation algorithms like 1st match, next fit, best fit, and worst match. currently in our formula we've tried our greatest to investigate the mistakes of those formulas and introduced a brand new algorithm referred to as TLSF (Two-level unintegrated Fit) memory authority which might be employed in Real Time package (RTOS). The TLSF formula provides express allocation and deallocation of memory blocks with a temporal value Θ(1).

Objective

The objective of the project are as follows: -

- To study general purpose OS and Realtime OS.
- To identify the problem in already discussed memory allocation algorithms for GP & RTOS.
- To introduce a new algorithm, for dynamic memory allocation that presents a bounded worst-case response time.
- To compare introduced algorithm with existing algorithms based on various parameters.

2. Introduction

The study of dynamic storage allocation (DSA) algorithms is a vital topic within the operational systems analysis and implementation areas and has been wide analyzed. conjointly with alternative basic computing issues, like looking and sorting, memory management is one amongst the foremost studied

and analyzed issues. thanks to the big quantity of existing DSA algorithms, we will get to assume that the matter of dynamic memory allocation has been already solved. this could be true for many application varieties, however true for time period applications is sort of completely different. In time period systems, it's required to grasp prior to the operation time bounds to research the system schedulable. The goal of DSA algorithms is to supply dynamically to the applications the quantity of memory needed at run time. In general, these algorithms ar designed to supply smart average response times, whereas Realtime applications can instead need the response times of the memory allocation and deallocation primitives to be finite. in addition, dynamic memory management will introduce a vital quantity of memory fragmentation, which will end in associate unreliable service once the applying runs for giant periods of your time. it's necessary to entails that a quick latent period and high turnout ar characteristics that require to be thought about in any quite system, and in time period systems. however, the most demand that defines these systems is to ensure the temporal arrangement constraints. Most DSA algorithms are designed to supply quick latent period for the foremost probable case, achieving an honest overall performance, though their worst-case latent period may be high or perhaps limitless. For these reasons, most RTOS developers and researchers avoid the utilization of dynamic memory in any respect, or use it in an exceedingly restricted method, as an example, solely throughout the system startup: “Developers of time period systems avoid the utilization of dynamic memory management as a result of they worry that the worst-case execution time of dynamic memory allocation routines isn't finite or is finite with a too necessary bound”. we've tried to introduce a brand new algorithmic rule, known as TLSF, for dynamic memory allocation that presents a finite worst-case latent period, whereas keeping the potency of the allocation and deallocation operations with a temporal value of Θ(1). additionally, the fragmentation drawback has higher impact in system performance with very long-time running applications. atiny low and finite fragmentation is additionally achieved by the projected algorithmic rule.

3. Literature Survey

Paper – [1]

Title: *Durgesh Raghuvanshi "Memory Management in Operating System"*

Methodology

Resources must be utilized efficiently to enhance performance. This paper demonstrates basic architecture of segmentation in an operating system and basic of its allocation. This paper also describes the basic concepts of virtual memory management and dynamic memory management.

PROS

This paper gives detailed overview of memory management schemes, paging and virtual memory.

CONS

There are not much negative in this paper, but it should have elaborated it much further in algo aspect.

Paper – [2]

Title: *Kumar, Dinesh & Singh, Mandeep. (2019). MEMORY MANAGEMENT IN OPERATING SYSTEM*

Methodology

This paper demonstrates the memory the executives in the working framework and it will show the fundamental design of division in a working framework and essentials of its assignment. This paper additionally portrays about the essential idea of the virtual memory the executives and the dynamic memory of the board.

PROS
This paper very well explains about memory addresses, swapping and memory allocation.

CONS
This paper gives very short explanations of some topics.

Paper – [3]

Title: *- Abdullah Awais, M. (2016). Memory Management: Challenges and Techniques for traditional Memory*

Methodology

There are different memory allocation algorithms had devised to organize memory efficiently in t = different timestamps according to the needs and scenario of usage yet there are issues and challenges of these allocators to provide full support for real time needs. These real time systems require memory on priority otherwise programs may crash or may be unresponsive if demanded memory is not allocated with the quick response. Besides the timing constraints, memory allocator algorithm must minimize the memory loss which comes in the form of fragmentation, the unused memory in response to the memory allocation needs because memory is allocated in the form of blocks. Our focus is to analyze traditional dynamic memory management algorithms with respect to their functionality, response time and efficiency to find out the issues and challenges with these allocators to sum up the knowledge to know the limitation of this algorithm which might reduce the performance of real time systems. This paper will give a comparative analysis of some well-known memory management techniques to highlight issues of real time systems and some innovative techniques suitable for these applications.

PROS
This article gives the detailed comparison of some known memory management system.

CONS
Some techniques have slow response time and larger allocation and deallocation time.

Paper – [4]

Title: *- Computer's Memory Management Research Paper*

Methodology

This paper tells us about the memory management in operating system and it will illustrate the basic types of operating system with its segmentation architecture and its allocation. Operating system doesn't care about the User's view of the process. It may divide the same function into different pages and those pages may or may not be loaded at the same time into the memory. It decreases the efficiency of the system.

It is better to have segmentation which divides the process into the segments. Each segment contain same type of functions such as main function can be included in one segment and the library functions can be included in the other segment.

PROS
This paper gives the detailed explanation of types of operating systems and segmentation.

CONS
This paper has very narrow domain.

Paper – [5]

Title: *- Miguel Masmano, Ismael Ripoll, Patricia Balbastre & Alfons Crespo. A constant-time dynamic storage allocator for real-time systems*

Methodology

Dynamic memory allocation has been used for decades. However, it has seldom been used in real-time systems since the worst case of spatial and temporal requirements for allocation and deallocation operations is either unbounded or bounded but with a very large bound.

PROS
In this paper, a new allocator called TLSF (Two Level Segregated Fit) is presented. TLSF is designed and implemented to accommodate real-time constraints. The proposed allocator exhibits time-bounded behavior, O (1), and maintains a very good execution time. This paper describes in detail the data structures and functions provided by TLSF. We also compare TLSF with a representative set of allocators regarding their temporal cost and fragmentation.

CONS
As this is an experimental paper so therefore no cons could me analyzed.

HARDWARE REQIREMENTS

CPU: Intel® Core™ CPUs higher than 6th gen

GPU: Not compulsory

Memory: 4GB or Higher

Storage: 250GB or Higher

SOFTWARE REQUIREMENTS

- gcc
- OS - Linux
- Command Prompt
- Vs Code

4. Proposed Methodology

4.1 Introduction

One of the key problems in systems is that the schedulability analysis to work out whether or not the system can satisfy the application’s temporal arrangement constraints at run-time or not. in spite of the analysis and planning techniques used, it's essential to work out the worst-case execution time (WCET) of all the running code, together with application code, library functions and package. Another characteristic that differentiates time period systems from different kinds of systems is that time period applications run giant periods of your time. Most non-real-time applications take solely many minutes or hours to finish their work and end. time period applications ar sometimes dead unendingly throughout the full lifetime of the system (months, years,). This behavior directly affects one in every of the vital aspects of dynamic memory management: the memory fragmentation downside. Considering of these aspects, the necessities of real time applications concerning dynamic memory is summarized as:
• finite reaction time. The worst-case execution time (WCET) of memory allocation and deallocation should be acknowledged prior to and be freelance of application information. this is often the most demand that has got to be met.
• quick reaction time. Besides having a finite reaction time, the reaction time should be short for the DSA algorithmic rule to be usable. A finite DSA algorithmic rule that's ten times slower than a standard one isn't helpful.
• Memory requests ought to be continually happy. Nonreal time applications will receive a null pointer or simply be killed by the OS once the system runs out of memory. though it's obvious that it's out of the question to continually grant all the memory requested, the DSA algorithmic rule must minimize the possibilities of exhausting the memory pool by minimizing the quantity of fragmentation and wasted memory.

With regard to memory needs, there exists an outsized vary of RTSs, from little embedded systems with little or no memory, no storage support (MMU) and no permanent storage; to giant, redundant, multi-processor systems. Our study and therefore the projected algorithmic rule have targeted on little systems wherever memory could be a scarce resource and there's no MMU support.
Regarding the means allocation and deallocation ar managed, we have a tendency to might realize 2 general approaches for DSA: (i) specific allocation and deallocation, wherever the applying has to expressly decision the primitives of the DSA algorithmic rule to assign memory (e.g., malloc) and to unharness it (e.g., free) once it's not required anymore; and (ii) implicit memory deallocation (also called Garbage Collection), wherever the DSA mechanism is responsible of collection the blocks of memory that are antecedently requested however aren't required any longer, so as to form them accessible for brand new allocation requests.

Besides temporal arrangement faults (due to associate limitless time DSA implementation) a memory allocation
may fail because of memory exhaustion. Memory exhaustion might occur because
of 2 reasons:

(i) The application needs additional memory that the whole memory accessible within
the system; or
(ii) The DSA algorithmic rule is unable to utilise memory that's free.

The DSA algorithmic rule will do nothing within the 1st case; the applying should be redesigned, or additional physical memory should be additional to the
system. traditionally the second reason has been thought-about as 2 completely different aspects referred to as internal and external fragmentation. Recent analysis considers these 2 fragmentation classes as one downside referred to as wasted memory or just fragmentation. Fragmentation has been thought-about a heavy issue. several initial experimental studies supported artificial or random workloads yield to the conclusion that the fragmentation downside might limit or perhaps stop the utilization of dynamic memory. different fascinating conclusions is:

• best-fit or physical first-fit policies turn out low fragmentation,
• blocks of memory ought to be now united
upon unharness, and
• ideally utilise the memory that has been recently discharged over those blocks that has been discharged more
in the past.

4.2 DSA Algorithms

The goal of DSA algorithms is to grant the applying the access to blocks of memory from a pool of free memory blocks. take issueent|the various} formula methods differ within the method they notice a free block of the foremost acceptable size. traditionally, DSA algorithms are classified in keeping with the subsequent categories:

• serial Fit: These algorithms area unit the foremost basic ones; they're supported one or double connected list of all free memory blocks. samples of algorithms supported this strategy area unit Fast-Fit, First-Fit, Next-Fit, and Worst-Fit. serial work isn't an
honest strategy for RTSs as a result of it's supported a serial search, whose value depends on the quantity of existing free blocks. This search will
be delimited however the certain isn't ordinarily acceptable.
• separate Free Lists: separate Free Lists algorithms use AN array of lists containing the references to free blocks of memory of a selected size (or within a selected size
range). once a block is deallocated, it's inserted within the listing similar to its
size. it's necessary to remark that the blocks area unit logically however not
physically separate. There area unit 2 variants of this
strategy: straightforward separate Storage and separate Fits.

Examples of this strategy area unit settler and Tadman's "Fast-Fit", that uses AN array of many, little size free lists and a binary tree for the free lists for larger sizes; and also the DSA developed by politician Lea. This strategy is suitable to be employed in RTOSs, since the looking out value isn't passionate about the quantity of free blocks.

• crony Systems: crony Systems area unit a variant of separate Free Lists, with economical split and merge operations. There area unit many variants of this technique, like Binary Buddies, Fibonacci Buddies, Weighted Buddies, and Double Buddies. crony Systems exhibit smart temporal arrangement behavior, adequate for RTOSs; however, they need the disadvantage of manufacturing giant internal fragmentation, of up to five hundredth [4].

• Indexed Fit: This strategy relies on the utilization of advanced structures to index the free memory blocks, with many fascinating options. samples of formulas mistreatment Indexed work area unit a Best-Fit algorithm employing a balanced tree, Stephenson's" Fast-Fit" distributor, that relies on a mathematician tree to store free blocks with 2 indexes, size, and memory address, etc. The Indexed work strategy will perform higher than separate Free Lists if the search time of a free block doesn't rely upon the quantity of free blocks.

• bitmap work: bitmap work ways area unit a variant of Indexed Fit ways that use a ikon to grasp that blocks area unit busy or free. AN example of this technique is that the Half-Fit formula [8]. This approach has the advantage with regard to the previous ones, that each one the data required to form the search is keep during a little piece of memory, generally thirty-two bits, thus reducing the chance of cache misses and up the interval.

5 . Architecture Methodology

5.1 Design Criteria of TLSF

There is not a single DSA algorithm suitable for all application types. As already stated, real-time applications requirements are quite different from conventional, general purpose applications. In this section we present the design of a new algorithm called Two-Level Segregated Fit (TLSF, for short) to fulfill the most important real-time requirements: a bounded and short response time, and a bounded and low fragmentation.

Moreover, the constraints that should be considered for embedded real-time systems are:

• Trusted environment: programmers that have access to the system are not malicious, that is, they will not try to intentionally steal or corrupt application data. The protection is done at the end-user interface level, not at the programming level.

• Small amount of physical memory available.

• No special hardware (MMU) available to support virtual memory.

To meet these constraints and requirements, TLSF has been designed with the following guidelines:

Immediate coalescing:

As soon as a block of memory is released, TLSF will immediately merge it with adjacent free blocks, if any, to build up a larger free block. Other DSA algorithms may defer coalescing, or even not coalesce at all. Deferred coalescing is a Useful strategy in systems where applications repeatedly use blocks of the same size. In such cases, deferred coalescing removes the overhead of continuously merging and splitting the same block.

Although deferred coalescing can improve the performance of the allocator, it adds unpredictability (a block request may require merging an unbounded number of free, but not merged blocks) and it also increases fragmentation. Therefore, deferred coalescing should not be used in real-time.

Splitting threshold:

The smallest block of allocatable memory is 16 bytes. Most applications, and real-time applications are not an exception, do not allocate simple data types like integers, pointers, or floating-point numbers, but more complex data structures which contains at least one pointer and a set of data. By limiting the minimum block size to 16 bytes, it is possible to store, inside the free blocks, the information needed to manage them, including the list of free blocks pointers. This approach optimizes the memory usage.

Good-fit strategy:

TLSF will try to return the smallest chunk of memory big enough to hold the requested block. Since most applications only use blocks of memory within a small range of sizes, a Best-Fit strategy tends to produce the lowest fragmentation on real workloads, compared to other approaches such as first-fit. A Best-Fit (or almost Best-Fit, also called Good-Fit) strategy can be implemented in an efficient and predictable manner by 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 sequence of requests, 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 free blocks whose size is between the size class (a range of sizes) and the next size class. Each segregated list contains blocks of the same class.

No reallocation:

We assume that the original memory pool is a single large block of free memory, and no sbrk()2 function is available. Since general purpose OSs (like UNIXr) provide virtual memory, most DSA algorithms were designed to take advantage of it. By using this capability, it is possible to design a DSA optimized to manage relatively small memory blocks, delegating the management of large blocks to the underlying OS by dynamically enlarging the memory pool. TLSF has been designed to provide complete support for dynamic memory, that is, it can be used by the applications and by the OS, using no virtual memory hardware.

Same strategy for all block sizes:

The same allocation strategy is used for any requested size. One of the most efficient and widely used DSA, Douglas Lea's allocator, uses four different strategies depending on the size of the request: First-Fit, cached blocks, segregated list and a system managing facility to enlarge the pool. This kind of algorithms do not provide a uniform behavior, so their worst-case execution time is usually high or even difficult to know. Memory is not cleaned-up: Neither the initial pool nor the free blocks are zeroed.

DSA algorithms used in multi-user environments must clean the memory (usually filling it with zeros) to avoid security problems. Not cleaning the memory before it is allocated to the user is considered a serious security flaw, because a malicious program could obtain confidential information.

5.2 TLSF Data Structures

The data structures used by the dynamic memory allocator proposed are described in this section. TLSF algorithm 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 speed up the access to the free blocks and also to manage a large set of segregated lists (to reduce fragmentation) the array of lists has been organized as a two-level array. The first-level array divides free blocks in classes that are a power of two apart (16, 32, 64, 128, etc.); and the second-level sub-divides each first-level class linearly, where the number of divisions (referred to as the Second Level Index, SLI) is a user configurable parameter. Each array of lists has an associated bitmap used to mark which lists are empty and which ones contain free blocks. Information regarding each block is stored inside the block itself.

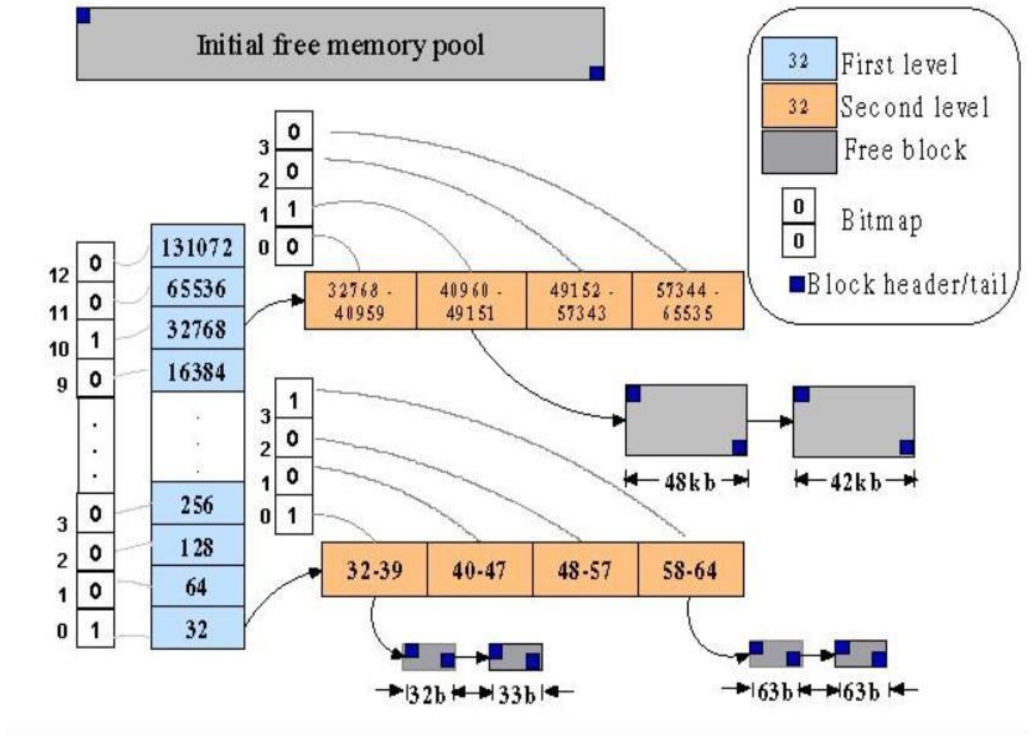
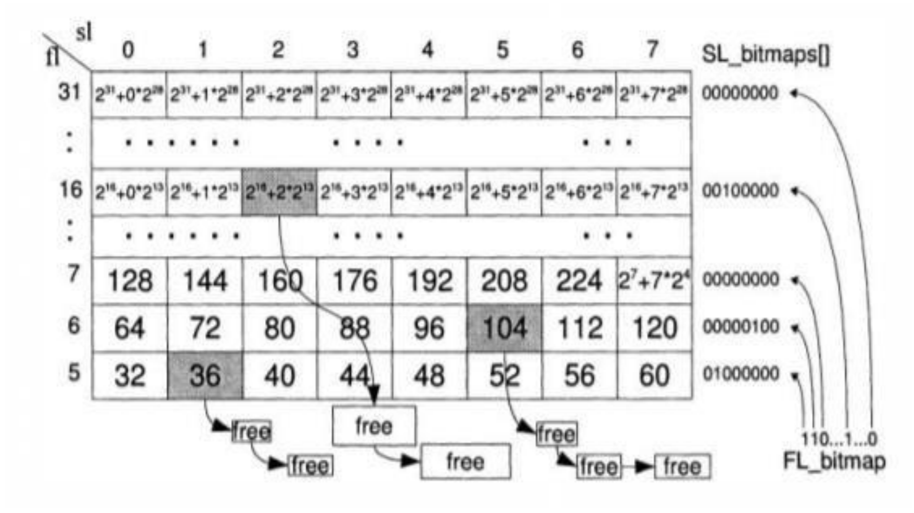
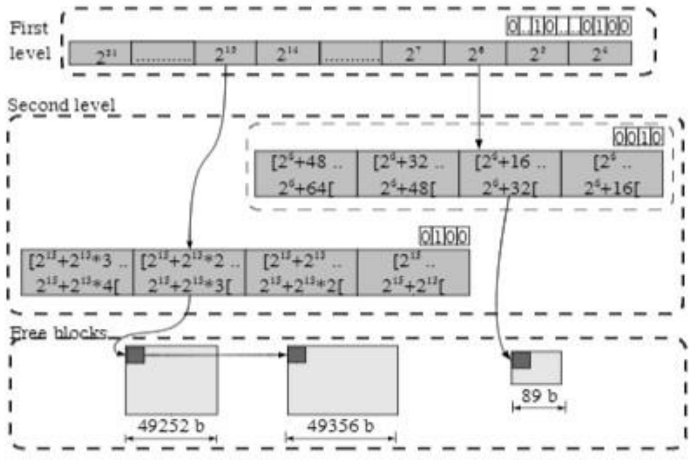


Figure 1





5.2.1 TLSF Block Header

The TLSF embeds into each block the information needed to manage the block (whether the block is free or not) and the pointers to link it into the two lists: the list of blocks of similar sizes and the list ordered by physical addresses. This data structure is called the block header. Since busy (used) blocks are not linked in any segregated list, their block header is smaller than the headers for free blocks.

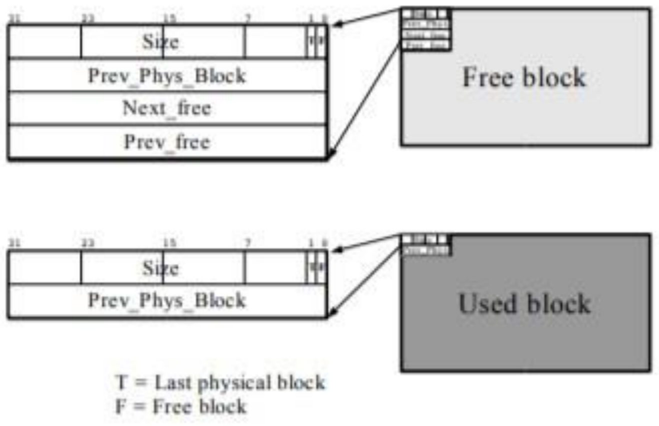


Figure 2. Free and allocated block headers.

The block header of free blocks contains the following data:

i) The size of the block, required to free the block and to link this block with the next one in the physical link list.

ii) Boundary tag, a pointer to the head of the previous physical block.

iii) Two pointers, to link this block into the corresponding segregated list (double linked list). The header of a busy block contains only the size and the boundary tag pointer. Since block sizes are always a multiple of four (the allocation unit is the word, 4 bytes) the two least significant bits of the size field are used to store the block status: block is busy or free (bit F), and whether the block is the last one of the pool or not (bit T).

5.2.2 TLSF Structure Function

Most TLSF operations think about the segregatelist() mapping operate. Given the scale of a block, the mapping operate calculates the indexes of the 2 arrays that time to the corresponding sequestered list that holds the requested block.

The operations provided by TLSF structure are:

• Initialise TLSF structure: This operate can produce the TLSF arrangement at the start of the pool (the remaining a part of the pool are the initial free memory). It accepts 3 parameters: a pointer to the memory pool, the scale of the pool, and also the second level index. it's attainable to form many freelance TLSF pools.

• Destroy TLSF structure: this operates marks the given pool as non-usable.

• Get a free block: Returns a pointer to a free space of memory of the requested size or larger. The requested size is rounded up to the closest list. This operation is enforced as follows:

First step: calculate the “f” and “s” indexes that area unit accustomed get the top of the list holding the nearer category (segregated) list. If this list isn't empty then the block at the top of the list is faraway from the list (marked as busy) and came back to the user; otherwise,
Second step: search succeeding (bigger than the requested size) nonempty sequestered list within the TSLF arrangement. This search is completed in constant time victimization the ikon masks and also the realize initial set (ffs) ikon instruction.
If an inventory is found, then the block at the top of the list are accustomed fulfill the request. Since this block is larger than requested, it's to be split and also the remaining is inserted into the corresponding sequestered list. On the opposite hand, if a list isn't found, then the request fails.

• Insert a free block: This operate inserts a free block within the TLSF structure. The mapping operate is employed to calculate the “f” and “s” indexes to seek out the sequestered list wherever the block must be inserted.

• Coalesce blocks: victimization the boundary tag technique, the top of the previous block is checked to check whether or not it's free or not. If the block is free, then the block is faraway from the sequestered list and incorporated with this block. a similar operation is meted out with succeeding physical block. Once the block has been incorporated with its free neighbor blocks, the large new block is inserted within the acceptable sequestered list.

6. Implementation

Coding Part:

- **tlsf.h**

```
#ifndef _TLSF_H_
#define _TLSF_H_
#include <sys/types.h>
extern size_t init_memory_pool(size_t, void *);
extern size_t get_used_size(void *);
extern size_t get_max_size(void *);
extern void destroy_memory_pool(void *);
extern size_t add_new_area(void *, size_t, void *);
extern void *malloc_ex(size_t, void *);
extern void free_ex(void *, void *);
extern void *realloc_ex(void *, size_t, void *);
extern void *calloc_ex(size_t, size_t, void *);
extern void *tlsf_malloc(size_t size);
extern void tlsf_free(void *ptr);

extern void *tlsf_realloc(void *ptr, size_t size);
extern void *tlsf_calloc(size_t nelem, size_t elem_size);
#endif
```

- **tlsf.c**

```
#ifndef USE_PRINTF
#define USE_PRINTF      (1)
#endif
#include <string.h>
#ifndef TLSF_USE_LOCKS
#define TLSF_USE_LOCKS  (0)
#endif
#ifndef TLSF_STATISTIC
#define TLSF_STATISTIC  (0)
#endif
#ifndef USE_MMAP
#define USE_MMAP    (0)
#endif
#ifndef USE_SBRK
#define USE_SBRK    (0)
#endif
#if TLSF_USE_LOCKS
#include "target.h"
#else
#define TLSF_CREATE_LOCK(_unused_)   do{}while(0)
#define TLSF_DESTROY_LOCK(_unused_)  do{}while(0)
#define TLSF_ACQUIRE_LOCK(_unused_)  do{}while(0)
#define TLSF_RELEASE_LOCK(_unused_)  do{}while(0)
#endif
#if TLSF_STATISTIC
#define TLSF_ADD_SIZE(tlsf, b) do {                                 \
        tlsf->used_size += (b->size & BLOCK_SIZE) + BHDR_OVERHEAD;  \
        if (tlsf->used_size > tlsf->max_size)                       \
            tlsf->max_size = tlsf->used_size;                       \
        } while(0)

#define TLSF_REMOVE_SIZE(tlsf, b) do {                              \
        tlsf->used_size -= (b->size & BLOCK_SIZE) + BHDR_OVERHEAD;  \
    } while(0)
#else
#define TLSF_ADD_SIZE(tlsf, b)       do{}while(0)
#define TLSF_REMOVE_SIZE(tlsf, b)    do{}while(0)
#endif
#if USE_MMAP || USE_SBRK
#include <unistd.h>
#endif
#if USE_MMAP
#include <sys/mman.h>
#endif
#include "tlsf.h"
```

```c
#if !defined(__GNUC__)
#ifndef __inline__
#define __inline__
#endif
#endif
/* The  debug functions  only can  be used  when _DEBUG_TLSF_  is set. */
#ifndef _DEBUG_TLSF_
#define _DEBUG_TLSF_  (0)
#endif
#define BLOCK_ALIGN (sizeof(void *) * 2)
#define MAX_FLI     (30)
#define MAX_LOG2_SLI    (5)
#define MAX_SLI     (1 << MAX_LOG2_SLI)     /* MAX_SLI = 2^MAX_LOG2_SLI */
#define FLI_OFFSET  (6)     /* tlsf structure just will manage blocks bigger */
/* than 128 bytes */
#define SMALL_BLOCK (128)
#define REAL_FLI    (MAX_FLI - FLI_OFFSET)
#define MIN_BLOCK_SIZE  (sizeof (free_ptr_t))
#define BHDR_OVERHEAD   (sizeof (bhdr_t) - MIN_BLOCK_SIZE)
#define TLSF_SIGNATURE  (0x2A59FA59)
#define PTR_MASK    (sizeof(void *) - 1)
#define BLOCK_SIZE  (0xFFFFFFFF - PTR_MASK)
#define GET_NEXT_BLOCK(_addr, _r) ((bhdr_t *) ((char *) (_addr) + (_r)))
#define MEM_ALIGN         ((BLOCK_ALIGN) - 1)
#define ROUNDUP_SIZE(_r)          (((_r) + MEM_ALIGN) & ~MEM_ALIGN)
#define ROUNDDOWN_SIZE(_r)        ((_r) & ~MEM_ALIGN)
#define ROUNDUP(_x, _v)           ((((~(_x)) + 1) & ((_v)-1)) + (_x))
#define BLOCK_STATE (0x1)
#define PREV_STATE  (0x2)
/* bit 0 of the block size */
#define FREE_BLOCK  (0x1)
#define USED_BLOCK  (0x0)
/* bit 1 of the block size */
#define PREV_FREE   (0x2)
#define PREV_USED   (0x0)
#define DEFAULT_AREA_SIZE (1024*10)
#ifdef USE_MMAP
#define PAGE_SIZE (getpagesize())
#endif
#ifdef USE_PRINTF
#include <stdio.h>
# define PRINT_MSG(fmt, args...) printf(fmt, ## args)
# define ERROR_MSG(fmt, args...) printf(fmt, ## args)
#else
# if !defined(PRINT_MSG)
#  define PRINT_MSG(fmt, args...)
# endif
# if !defined(ERROR_MSG)
#  define ERROR_MSG(fmt, args...)
# endif
#endif
typedef unsigned int u32_t;     /* NOTE: Make sure that this type is 4 bytes long on your computer */
typedef unsigned char u8_t;     /* NOTE: Make sure that this type is 1 byte on your computer */
typedef struct free_ptr_struct {
    struct bhdr_struct *prev;
    struct bhdr_struct *next;
} free_ptr_t;
typedef struct bhdr_struct {
    /* This pointer is just valid if the first bit of size is set */
    struct bhdr_struct *prev_hdr;
    /* The size is stored in bytes */
    size_t size;                /* bit 0 indicates whether the block is used and */
    /* bit 1 allows to know whether the previous block is free */
    union {
        struct free_ptr_struct free_ptr;
        u8_t buffer[1];         /*sizeof(struct free_ptr_struct)]; */
    } ptr;
} bhdr_t;
typedef struct area_info_struct {
    bhdr_t *end;
    struct area_info_struct *next;
} area_info_t;
typedef struct TLSF_struct {
    /* the TLSF's structure signature */
    u32_t tlsf_signature;
#if TLSF_USE_LOCKS
    TLSF_MLOCK_T lock;
#endif
#if TLSF_STATISTIC
    /* These can not be calculated outside tlsf because we
     * do not know the sizes when freeing/reallocing memory. */
    size_t used_size;
    size_t max_size;
#endif
    /* A linked list holding all the existing areas */
    area_info_t *area_head;
    u32_t fl_bitmap;
    u32_t sl_bitmap[REAL_FLI];
    bhdr_t *matrix[REAL_FLI][MAX_SLI];
} tlsf_t;
static __inline__ void set_bit(int nr, u32_t * addr);
static __inline__ void clear_bit(int nr, u32_t * addr);
static __inline__ int ls_bit(int x);
static __inline__ int ms_bit(int x);
static __inline__ void MAPPING_SEARCH(size_t * _r, int *_fl, int *_sl);
static __inline__ void MAPPING_INSERT(size_t _r, int *_fl, int *_sl);
static __inline__ bhdr_t *FIND_SUITABLE_BLOCK(tlsf_t * _tlsf, int *_fl, int *_sl);
static __inline__ bhdr_t *process_area(void *area, size_t size);
```

```
#if USE_SBRK || USE_MMAP
static __inline__ void *get_new_area(size_t * size);
#endif
static const int table[] = {
    -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4,
    4, 4,
    4, 4, 4, 4, 4, 4, 4,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    5,
    5, 5, 5, 5, 5, 5, 5,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6,
    6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6,
    6, 6, 6, 6, 6, 6, 6,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7,
    7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7,
    7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7,
    7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7,
    7, 7, 7, 7, 7, 7, 7
};
static __inline__ int ls_bit(int i)
{
    unsigned int a;
    unsigned int x = i & -i;
    a = x <= 0xffff ? (x <= 0xff ? 0 : 8) : (x <= 0xffffff ? 16 : 24);
    return table[x >> a] + a;
}
static __inline__ int ms_bit(int i)
{
    unsigned int a;
    unsigned int x = (unsigned int) i;
    a = x <= 0xffff ? (x <= 0xff ? 0 : 8) : (x <= 0xffffff ? 16 : 24);
    return table[x >> a] + a;
}
static __inline__ void set_bit(int nr, u32_t * addr)
{
    addr[nr >> 5] |= 1 << (nr & 0x1f);
}
static __inline__ void clear_bit(int nr, u32_t * addr)
{
    addr[nr >> 5] &= ~(1 << (nr & 0x1f));
}
static __inline__ void MAPPING_SEARCH(size_t * _r, int *_fl, int *_sl)
{
    int _t;
    if (*_r < SMALL_BLOCK) {
        *_fl = 0;
        *_sl = *_r / (SMALL_BLOCK / MAX_SLI);
    } else {
        _t = (1 << (ms_bit(*_r) - MAX_LOG2_SLI)) - 1;
        *_r = *_r + _t;
        *_fl = ms_bit(*_r);
        *_sl = (*_r >> (*_fl - MAX_LOG2_SLI)) - MAX_SLI;
        *_fl -= FLI_OFFSET;
        *_r &= ~_t;
    }
}
static __inline__ void MAPPING_INSERT(size_t _r, int *_fl, int *_sl)
{
    if (_r < SMALL_BLOCK) {
        *_fl = 0;
        *_sl = _r / (SMALL_BLOCK / MAX_SLI);
    } else {
        *_fl = ms_bit(_r);
        *_sl = (_r >> (*_fl - MAX_LOG2_SLI)) - MAX_SLI;
        *_fl -= FLI_OFFSET;
    }
}
static __inline__ bhdr_t *FIND_SUITABLE_BLOCK(tlsf_t * _tlsf, int *_fl, int *_sl)
{
    u32_t _tmp = _tlsf->sl_bitmap[*_fl] & (~0 << *_sl);
    bhdr_t *_b = NULL;
    if (_tmp) {
        *_sl = ls_bit(_tmp);
        _b = _tlsf->matrix[*_fl][*_sl];
    } else {
        *_fl = ls_bit(_tlsf->fl_bitmap & (~0 << (*_fl + 1)));
        if (*_fl > 0) {         /* likely */
            *_sl = ls_bit(_tlsf->sl_bitmap[*_fl]);
            _b = _tlsf->matrix[*_fl][*_sl];
        }
    }
    return _b;
}
#define EXTRACT_BLOCK_HDR(_b, _tlsf, _fl, _sl) do {                    \
        _tlsf -> matrix [_fl] [_sl] = _b -> ptr.free_ptr.next;      \
        if (_tlsf -> matrix[_fl][_sl])                              \
            _tlsf -> matrix[_fl][_sl] -> ptr.free_ptr.prev = NULL;  \
        else {                                                      \
            clear_bit (_sl, &_tlsf -> sl_bitmap [_fl]);             \
            if (!_tlsf -> sl_bitmap [_fl])                          \
                clear_bit (_fl, &_tlsf -> fl_bitmap);               \
        }                                                           \
        _b -> ptr.free_ptr.prev =  NULL;                \
        _b -> ptr.free_ptr.next =  NULL;                \
    }while(0)
#define EXTRACT_BLOCK(_b, _tlsf, _fl, _sl) do {                         \
        if (_b -> ptr.free_ptr.next)                                    \
            _b -> ptr.free_ptr.next -> ptr.free_ptr.prev = _b -> ptr.free_ptr.prev; \
        if (_b -> ptr.free_ptr.prev)                                    \
            _b -> ptr.free_ptr.prev -> ptr.free_ptr.next = _b -> ptr.free_ptr.next; \
        if (_tlsf -> matrix [_fl][_sl] == _b) {                         \
            _tlsf -> matrix [_fl][_sl] = _b -> ptr.free_ptr.next;       \
            if (!_tlsf -> matrix [_fl][_sl]) {                          \
                clear_bit (_sl, &_tlsf -> sl_bitmap[_fl]);              \
                if (!_tlsf -> sl_bitmap [_fl])                          \
                    clear_bit (_fl, &_tlsf -> fl_bitmap);               \
            }                                                           \
        }                                                               \
        _b -> ptr.free_ptr.prev = NULL;                 \
        _b -> ptr.free_ptr.next = NULL;                 \
    } while(0)

#define INSERT_BLOCK(_b, _tlsf, _fl, _sl) do {                          \
        _b -> ptr.free_ptr.prev = NULL; \
        _b -> ptr.free_ptr.next = _tlsf -> matrix [_fl][_sl]; \
        if (_tlsf -> matrix [_fl][_sl])                                 \
            _tlsf -> matrix [_fl][_sl] -> ptr.free_ptr.prev = _b;       \
        _tlsf -> matrix [_fl][_sl] = _b;                                \
        set_bit (_sl, &_tlsf -> sl_bitmap [_fl]);                       \
        set_bit (_fl, &_tlsf -> fl_bitmap);                             \
    } while(0)
#if USE_SBRK || USE_MMAP
static __inline__ void *get_new_area(size_t * size)
{
    void *area;
#if USE_SBRK
    area = (void *)sbrk(0);
    if (((void *)sbrk(*size)) != ((void *) -1))
        return area;
#endif
#ifndef MAP_ANONYMOUS
/* https://dev.openwrt.org/ticket/322 */
# define MAP_ANONYMOUS MAP_ANON
#endif
#if USE_MMAP
    *size = ROUNDUP(*size, PAGE_SIZE);
    if ((area = mmap(0, *size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -
1, 0)) != MAP_FAILED)
        return area;
#endif
    return ((void *) ~0);
}
#endif
static __inline__ bhdr_t *process_area(void *area, size_t size)
{
    bhdr_t *b, *lb, *ib;
    area_info_t *ai;
    ib = (bhdr_t *) area;
    ib->size =
        (sizeof(area_info_t) <
         MIN_BLOCK_SIZE) ? MIN_BLOCK_SIZE : ROUNDUP_SIZE(sizeof(area_info_t)) | USED_BLOCK | PREV_USED;
    b = (bhdr_t *) GET_NEXT_BLOCK(ib->ptr.buffer, ib->size & BLOCK_SIZE);
    b->size = ROUNDDOWN_SIZE(size - 3 * BHDR_OVERHEAD - (ib-
>size & BLOCK_SIZE)) | USED_BLOCK | PREV_USED;
    b->ptr.free_ptr.prev = b->ptr.free_ptr.next = 0;
    lb = GET_NEXT_BLOCK(b->ptr.buffer, b->size & BLOCK_SIZE);
    lb->prev_hdr = b;
    lb->size = 0 | USED_BLOCK | PREV_FREE;
    ai = (area_info_t *) ib->ptr.buffer;
    ai->next = 0;
    ai->end = lb;
    return ib;
}

/******************************************************************/
/******************** Begin of the allocator code ****************/
/******************************************************************/

static char *mp = NULL;         /* Default memory pool. */

/******************************************************************/
size_t init_memory_pool(size_t mem_pool_size, void *mem_pool)
{
/******************************************************************/
    tlsf_t *tlsf;
    bhdr_t *b, *ib;
    if (!mem_pool || !mem_pool_size || mem_pool_size < sizeof(tlsf_t) + BHDR_OVERHEAD * 8) {
        ERROR_MSG("init_memory_pool (): memory_pool invalid\n");
        return -1;
    }
    if (((unsigned long) mem_pool & PTR_MASK)) {
        ERROR_MSG("init_memory_pool (): mem_pool must be aligned to a word\n");
        return -1;
    }
    tlsf = (tlsf_t *) mem_pool;
```

```c
    /* Check if already initialised */
    if (tlsf->tlsf_signature == TLSF_SIGNATURE) {
        mp = mem_pool;
        b = GET_NEXT_BLOCK(mp, ROUNDUP_SIZE(sizeof(tlsf_t)));
        return b->size & BLOCK_SIZE;
    }
    mp = mem_pool;
/* Zeroing the memory pool */
    memset(mem_pool, 0, sizeof(tlsf_t));
    tlsf->tlsf_signature = TLSF_SIGNATURE;
    TLSF_CREATE_LOCK(&tlsf->lock);
    ib = process_area(GET_NEXT_BLOCK
        (mem_pool, ROUNDUP_SIZE(sizeof(tlsf_t))), ROUNDDOWN_SIZE(mem_pool_size - sizeof(tlsf_t)));
    b = GET_NEXT_BLOCK(ib->ptr.buffer, ib->size & BLOCK_SIZE);
    free_ex(b->ptr.buffer, tlsf);
    tlsf->area_head = (area_info_t *) ib->ptr.buffer;
#if TLSF_STATISTIC
    tlsf->used_size = mem_pool_size - (b->size & BLOCK_SIZE);
    tlsf->max_size = tlsf->used_size;
#endif
    return (b->size & BLOCK_SIZE);
}
/******************************************************************/
size_t add_new_area(void *area, size_t area_size, void *mem_pool)
{
/******************************************************************/
    tlsf_t *tlsf = (tlsf_t *) mem_pool;
    area_info_t *ptr, *ptr_prev, *ai;
    bhdr_t *ib0, *b0, *lb0, *ib1, *b1, *lb1, *next_b;
    memset(area, 0, area_size);
    ptr = tlsf->area_head;
    ptr_prev = 0;
    ib0 = process_area(area, area_size);
    b0 = GET_NEXT_BLOCK(ib0->ptr.buffer, ib0->size & BLOCK_SIZE);
    lb0 = GET_NEXT_BLOCK(b0->ptr.buffer, b0->size & BLOCK_SIZE);

    /* Before inserting the new area, we have to merge this area with the
       already existing ones */

    while (ptr) {
        ib1 = (bhdr_t *) ((char *) ptr - BHDR_OVERHEAD);
        b1 = GET_NEXT_BLOCK(ib1->ptr.buffer, ib1->size & BLOCK_SIZE);
        lb1 = ptr->end;
        /* Merging the new area with the next physically contigous one */
        if ((unsigned long) ib1 == (unsigned long) lb0 + BHDR_OVERHEAD) {
            if (tlsf->area_head == ptr) {
                tlsf->area_head = ptr->next;
                ptr = ptr->next;
            } else {
                ptr_prev->next = ptr->next;
                ptr = ptr->next;
            }
            b0->size =
                ROUNDDOWN_SIZE((b0->size & BLOCK_SIZE) +
                               (ib1->size & BLOCK_SIZE) + 2 * BHDR_OVERHEAD) | USED_BLOCK | PREV_USED;
            b1->prev_hdr = b0;
            lb0 = lb1;
            continue;
        }
        /* Merging the new area with the previous physically contigous
           one */
        if ((unsigned long) lb1->ptr.buffer == (unsigned long) ib0) {
            if (tlsf->area_head == ptr) {
                tlsf->area_head = ptr->next;
                ptr = ptr->next;
            } else {
                ptr_prev->next = ptr->next;
                ptr = ptr->next;
            }
            lb1->size =
                ROUNDDOWN_SIZE((b0->size & BLOCK_SIZE) +
               (ib0->size & BLOCK_SIZE) + 2 * BHDR_OVERHEAD) | USED_BLOCK | (lb1>size & PREV_STATE);
            next_b = GET_NEXT_BLOCK(lb1->ptr.buffer, lb1->size & BLOCK_SIZE);
            next_b->prev_hdr = lb1;
            b0 = lb1;
            ib0 = ib1;
            continue;
        }
        ptr_prev = ptr;
        ptr = ptr->next;
    }
    /* Inserting the area in the list of linked areas */
    ai = (area_info_t *) ib0->ptr.buffer;
    ai->next = tlsf->area_head;
    ai->end = lb0;
    tlsf->area_head = ai;
    free_ex(b0->ptr.buffer, mem_pool);
    return (b0->size & BLOCK_SIZE);
}
size_t get_used_size(void *mem_pool)
{
/******************************************************************/
#if TLSF_STATISTIC
    return ((tlsf_t *) mem_pool)->used_size;
#else
    return 0;
#endif
}
```

```
size_t get_max_size(void *mem_pool)
{
#if TLSF_STATISTIC
    return ((tlsf_t *) mem_pool)->max_size;
#else
    return 0;
#endif
}
void destroy_memory_pool(void *mem_pool)
{
    tlsf_t *tlsf = (tlsf_t *) mem_pool;
    tlsf->tlsf_signature = 0;
    TLSF_DESTROY_LOCK(&tlsf->lock);

}
void *tlsf_malloc(size_t size)
{void *ret;
#if USE_MMAP || USE_SBRK
    if (!mp) {size_t area_size;

void *area;
        area_size = sizeof(tlsf_t) + BHDR_OVERHEAD * 8; /* Just a safety constant */
        area_size = (area_size > DEFAULT_AREA_SIZE) ? area_size : DEFAULT_AREA_SIZE;
        area = get_new_area(&area_size);
        if (area == ((void *) ~0))
            return NULL;        /* Not enough system memory */
        init_memory_pool(area_size, area);
    }
#endif
    TLSF_ACQUIRE_LOCK(&((tlsf_t *)mp)->lock);
    ret = malloc_ex(size, mp);
    TLSF_RELEASE_LOCK(&((tlsf_t *)mp)->lock);
    return ret;
}

void tlsf_free(void *ptr)
{
    TLSF_ACQUIRE_LOCK(&((tlsf_t *)mp)->lock);
    free_ex(ptr, mp);
    TLSF_RELEASE_LOCK(&((tlsf_t *)mp)->lock);
}
void *tlsf_realloc(void *ptr, size_t size)
{    void *ret;
#if USE_MMAP || USE_SBRK
    if (!mp) {
        return tlsf_malloc(size);
    }
#endif
    TLSF_ACQUIRE_LOCK(&((tlsf_t *)mp)->lock);
    ret = realloc_ex(ptr, size, mp);
    TLSF_RELEASE_LOCK(&((tlsf_t *)mp)->lock);
    return ret;
}
void *tlsf_calloc(size_t nelem, size_t elem_size)
{
    void *ret;
    TLSF_ACQUIRE_LOCK(&((tlsf_t *)mp)->lock);
    ret = calloc_ex(nelem, elem_size, mp);
    TLSF_RELEASE_LOCK(&((tlsf_t *)mp)->lock);
    return ret;
}
void *malloc_ex(size_t size, void *mem_pool)
{
    tlsf_t *tlsf = (tlsf_t *) mem_pool;
    bhdr_t *b, *b2, *next_b;
    int fl, sl;
    size_t tmp_size;
    size = (size < MIN_BLOCK_SIZE) ? MIN_BLOCK_SIZE : ROUNDUP_SIZE(size);

    MAPPING_SEARCH(&size, &fl, &sl);

    /* Searching a free block, recall that this function changes the values of fl and sl,
       so they are not longer valid when the function fails */
    b = FIND_SUITABLE_BLOCK(tlsf, &fl, &sl);
#if USE_MMAP || USE_SBRK
    if (!b) {
        size_t area_size;
        void *area;
        /* Growing the pool size when needed */
        area_size = size + BHDR_OVERHEAD * 8;   /* size plus enough room for the requered headers. */
        area_size = (area_size > DEFAULT_AREA_SIZE) ? area_size : DEFAULT_AREA_SIZE;
        area = get_new_area(&area_size);        /* Call sbrk or mmap */
        if (area == ((void *) ~0))
            return NULL;        /* Not enough system memory */
        add_new_area(area, area_size, mem_pool);
        /* Rounding up the requested size and calculating fl and sl */
        MAPPING_SEARCH(&size, &fl, &sl);
        /* Searching a free block */
        b = FIND_SUITABLE_BLOCK(tlsf, &fl, &sl);
    }
#endif
    if (!b)
        return NULL;            /* Not found */

    EXTRACT_BLOCK_HDR(b, tlsf, fl, sl);
    /*-- found: */
    next_b = GET_NEXT_BLOCK(b->ptr.buffer, b->size & BLOCK_SIZE);
    /* Should the block be split? */
    tmp_size = (b->size & BLOCK_SIZE) - size;
    if (tmp_size >= sizeof(bhdr_t)) {
        tmp_size -= BHDR_OVERHEAD;
        b2 = GET_NEXT_BLOCK(b->ptr.buffer, size);
        b2->size = tmp_size | FREE_BLOCK | PREV_USED;
        next_b->prev_hdr = b2;
        MAPPING_INSERT(tmp_size, &fl, &sl);
        INSERT_BLOCK(b2, tlsf, fl, sl);
        b->size = size | (b->size & PREV_STATE);
    } else {
        next_b->size &= (~PREV_FREE);
        b->size &= (~FREE_BLOCK);       /* Now it's used */
    }
    TLSF_ADD_SIZE(tlsf, b);
    return (void *) b->ptr.buffer;
}

void free_ex(void *ptr, void *mem_pool)
{
/******************************************************************/
    tlsf_t *tlsf = (tlsf_t *) mem_pool;
    bhdr_t *b, *tmp_b;
    int fl = 0, sl = 0;
    if (!ptr) {
        return;
    }
    b = (bhdr_t *) ((char *) ptr - BHDR_OVERHEAD);
    b->size |= FREE_BLOCK;
    TLSF_REMOVE_SIZE(tlsf, b);
    b->ptr.free_ptr.prev = NULL;
    b->ptr.free_ptr.next = NULL;
    tmp_b = GET_NEXT_BLOCK(b->ptr.buffer, b->size & BLOCK_SIZE);
    if (tmp_b->size & FREE_BLOCK) {
        MAPPING_INSERT(tmp_b->size & BLOCK_SIZE, &fl, &sl);
        EXTRACT_BLOCK(tmp_b, tlsf, fl, sl);
        b->size += (tmp_b->size & BLOCK_SIZE) + BHDR_OVERHEAD;
    }
    if (b->size & PREV_FREE) {
        tmp_b = b->prev_hdr;
        MAPPING_INSERT(tmp_b->size & BLOCK_SIZE, &fl, &sl);
        EXTRACT_BLOCK(tmp_b, tlsf, fl, sl);
        tmp_b->size += (b->size & BLOCK_SIZE) + BHDR_OVERHEAD;
        b = tmp_b;
    }
    MAPPING_INSERT(b->size & BLOCK_SIZE, &fl, &sl);
    INSERT_BLOCK(b, tlsf, fl, sl);
    tmp_b = GET_NEXT_BLOCK(b->ptr.buffer, b->size & BLOCK_SIZE);
    tmp_b->size |= PREV_FREE;
    tmp_b->prev_hdr = b;
```

```
}
void *realloc_ex(void *ptr, size_t new_size, void *mem_pool)
{    tlsf_t *tlsf = (tlsf_t *) mem_pool;
    void *ptr_aux;
    unsigned int cpsize;
    bhdr_t *b, *tmp_b, *next_b;
    int fl, sl;
    size_t tmp_size;
    if (!ptr) {
        if (new_size)
            return (void *) malloc_ex(new_size, mem_pool);
        if (!new_size)
            return NULL;
    } else if (!new_size) {
        free_ex(ptr, mem_pool);
        return NULL;
    }
    b = (bhdr_t *) ((char *) ptr - BHDR_OVERHEAD);
    next_b = GET_NEXT_BLOCK(b->ptr.buffer, b->size & BLOCK_SIZE);
    new_size = (new_size < MIN_BLOCK_SIZE) ? MIN_BLOCK_SIZE : ROUNDUP_SIZE(new_size);
    tmp_size = (b->size & BLOCK_SIZE);
    if (new_size <= tmp_size) {
    TLSF_REMOVE_SIZE(tlsf, b);
        if (next_b->size & FREE_BLOCK) {
            MAPPING_INSERT(next_b->size & BLOCK_SIZE, &fl, &sl);
            EXTRACT_BLOCK(next_b, tlsf, fl, sl);
            tmp_size += (next_b->size & BLOCK_SIZE) + BHDR_OVERHEAD;
            next_b = GET_NEXT_BLOCK(next_b->ptr.buffer, next_b->size & BLOCK_SIZE);
        }
        tmp_size -= new_size;
        if (tmp_size >= sizeof(bhdr_t)) {
            tmp_size -= BHDR_OVERHEAD;
            tmp_b = GET_NEXT_BLOCK(b->ptr.buffer, new_size);
            tmp_b->size = tmp_size | FREE_BLOCK | PREV_USED;
            next_b->prev_hdr = tmp_b;
            next_b->size |= PREV_FREE;
            MAPPING_INSERT(tmp_size, &fl, &sl);
            INSERT_BLOCK(tmp_b, tlsf, fl, sl);
            b->size = new_size | (b->size & PREV_STATE);
        }
    TLSF_ADD_SIZE(tlsf, b);
        return (void *) b->ptr.buffer;
    }
    if ((next_b->size & FREE_BLOCK)) {
        if (new_size <= (tmp_size + (next_b->size & BLOCK_SIZE))) {
            TLSF_REMOVE_SIZE(tlsf, b);
            MAPPING_INSERT(next_b->size & BLOCK_SIZE, &fl, &sl);
            EXTRACT_BLOCK(next_b, tlsf, fl, sl);
            b->size += (next_b->size & BLOCK_SIZE) + BHDR_OVERHEAD;
```

```c
            next_b = GET_NEXT_BLOCK(b->ptr.buffer, b->size & BLOCK_SIZE);
            next_b->prev_hdr = b;
            next_b->size &= ~PREV_FREE;
            tmp_size = (b->size & BLOCK_SIZE) - new_size;
            if (tmp_size >= sizeof(bhdr_t)) {
                tmp_size -= BHDR_OVERHEAD;
                tmp_b = GET_NEXT_BLOCK(b->ptr.buffer, new_size);
                tmp_b->size = tmp_size | FREE_BLOCK | PREV_USED;
                next_b->prev_hdr = tmp_b;
                next_b->size |= PREV_FREE;
                MAPPING_INSERT(tmp_size, &fl, &sl);
                INSERT_BLOCK(tmp_b, tlsf, fl, sl);
                b->size = new_size | (b->size & PREV_STATE);
            }
            TLSF_ADD_SIZE(tlsf, b);
            return (void *) b->ptr.buffer;
        }
    }
    if (!(ptr_aux = malloc_ex(new_size, mem_pool))){
        return NULL;
    }
    cpsize = ((b->size & BLOCK_SIZE) > new_size) ? new_size : (b->size & BLOCK_SIZE);
    memcpy(ptr_aux, ptr, cpsize);
    free_ex(ptr, mem_pool);
    return ptr_aux;
}
void *calloc_ex(size_t nelem, size_t elem_size, void *mem_pool)
{
    void *ptr;
    if (nelem <= 0 || elem_size <= 0)
        return NULL;
    if (!(ptr = malloc_ex(nelem * elem_size, mem_pool)))
        return NULL;
    memset(ptr, 0, nelem * elem_size);
    return ptr;
}
#if _DEBUG_TLSF_
extern void dump_memory_region(unsigned char *mem_ptr, unsigned int size);
extern void print_block(bhdr_t * b);
extern void print_tlsf(tlsf_t * tlsf);
void print_all_blocks(tlsf_t * tlsf);
void dump_memory_region(unsigned char *mem_ptr, unsigned int size)
{
    unsigned long begin = (unsigned long) mem_ptr;
    unsigned long end = (unsigned long) mem_ptr + size;
    int column = 0;
    begin >>= 2;
    begin <<= 2;
    end >>= 2;
    end++;
    end <<= 2;
    PRINT_MSG("\nMemory region dumped: 0x%lx - 0x%lx\n\n", begin, end);
    column = 0;
    PRINT_MSG("0x%lx ", begin);
    while (begin < end) {
        if (((unsigned char *) begin)[0] == 0)
            PRINT_MSG("00");
        else
            PRINT_MSG("%02x", ((unsigned char *) begin)[0]);
        if (((unsigned char *) begin)[1] == 0)
            PRINT_MSG("00 ");
        else
            PRINT_MSG("%02x ", ((unsigned char *) begin)[1]);
        begin += 2;
        column++;
        if (column == 8) {
            PRINT_MSG("\n0x%lx ", begin);
            column = 0;
        }
    }
    PRINT_MSG("\n\n");
}
void print_block(bhdr_t * b)
{
    if (!b)
        return;
    PRINT_MSG(">> [%p] (", b);
    if ((b->size & BLOCK_SIZE))
        PRINT_MSG("%lu bytes, ", (unsigned long) (b->size & BLOCK_SIZE));
    else
        PRINT_MSG("sentinel, ");
    if ((b->size & BLOCK_STATE) == FREE_BLOCK)
        PRINT_MSG("free [%p, %p], ", b->ptr.free_ptr.prev, b->ptr.free_ptr.next);
    else
        PRINT_MSG("used, ");
    if ((b->size & PREV_STATE) == PREV_FREE)
        PRINT_MSG("prev. free [%p])\n", b->prev_hdr);
    else
        PRINT_MSG("prev used)\n");
}
void print_tlsf(tlsf_t * tlsf)
{
    bhdr_t *next;
    int i, j;
    PRINT_MSG("\nTLSF at %p\n", tlsf);
    PRINT_MSG("FL bitmap: 0x%x\n\n", (unsigned) tlsf->fl_bitmap);
    for (i = 0; i < REAL_FLI; i++) {
        if (tlsf->sl_bitmap[i])
```

```c
            PRINT_MSG("SL bitmap 0x%x\n", (unsigned) tlsf->sl_bitmap[i]);
        for (j = 0; j < MAX_SLI; j++) {
            next = tlsf->matrix[i][j];
            if (next)
                PRINT_MSG("-> [%d][%d]\n", i, j);
            while (next) {
                print_block(next);
                next = next->ptr.free_ptr.next;
            }
        }
    }
}

void print_all_blocks(tlsf_t * tlsf)
{
    area_info_t *ai;
    bhdr_t *next;
    PRINT_MSG("\nTLSF at %p\nALL BLOCKS\n\n", tlsf);
    ai = tlsf->area_head;
    while (ai) {
        next = (bhdr_t *) ((char *) ai - BHDR_OVERHEAD);
        while (next) {
            print_block(next);
            if ((next->size & BLOCK_SIZE))
                next = GET_NEXT_BLOCK(next->ptr.buffer, next->size & BLOCK_SIZE);
            else
                next = NULL;
        }
        ai = ai->next;
    }
}

#endif
```

We will be using a ***Makefile*** which will create an object file for each (.c) extension code to compile all the codes at the same time when testing using our test function ***(test1.c).***

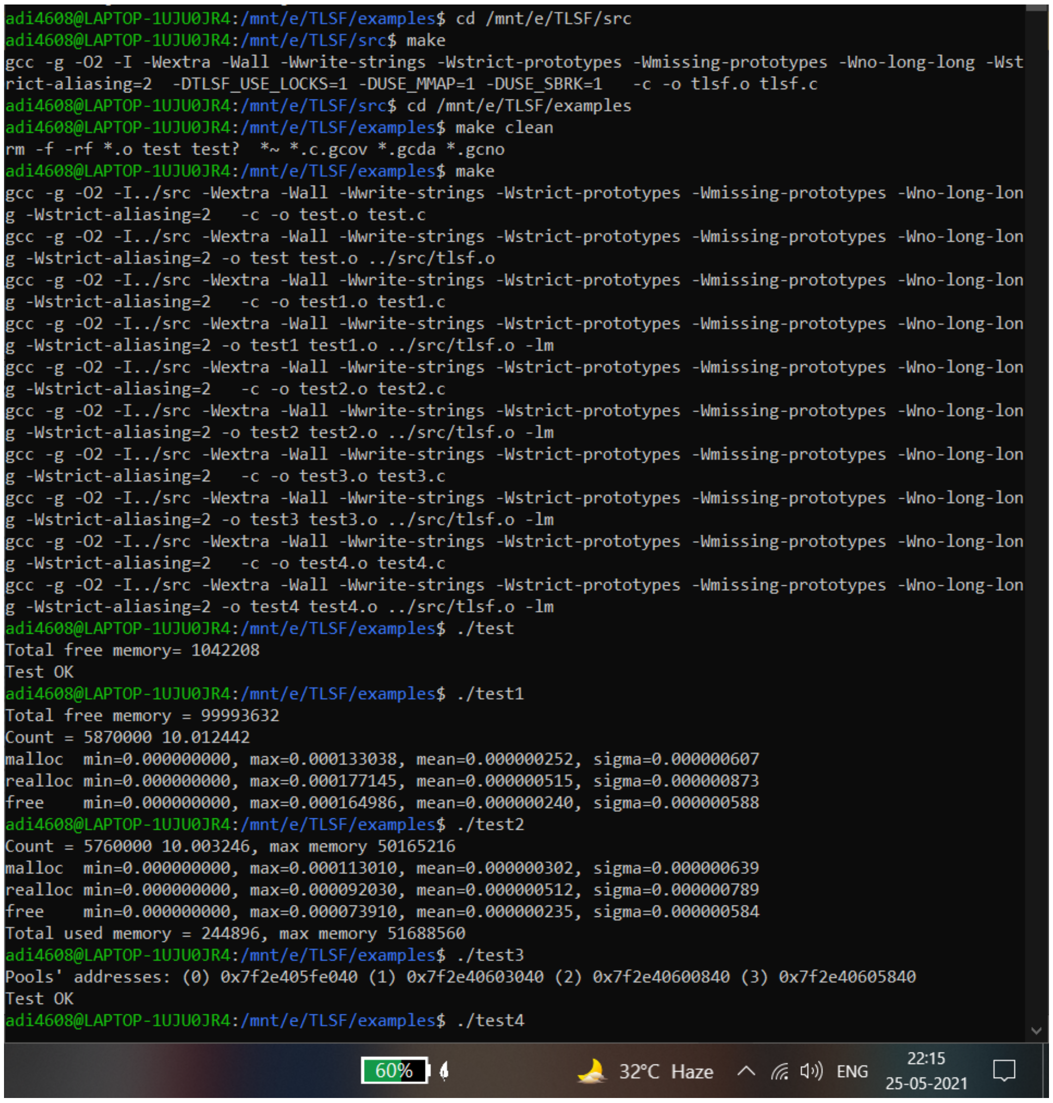
- **test1.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <limits.h>
#include <string.h>
#include "tlsf.h"
#define NUM_MALLOC      (10000)
#define SIZE_MALLOC     (10000)
#define POOL_SIZE (NUM_MALLOC*SIZE_MALLOC)
#define FILL_MEM  (0)
typedef struct
{
  const char *name;
  double min;
  double max;
  double n;
  double sum;
  double sumquad;
} minmax;

static double
getcurtime (void)
{
  struct timeval tv;
  struct timezone tz;

  gettimeofday (&tv, &tz);
  return ((tv.tv_usec / 1000000.0) + (1.0 * tv.tv_sec));
}

static void
init (minmax * data)
{
  data->name = "";
  data->min = 1e37;
  data->max = 0.0;
  data->n = 0.0;
  data->sum = 0.0;
  data->sumquad = 0.0;
}
static void
update (int print, const char *n, double s, double e, minmax * data)
{
  double d = e - s;
  double mean, sigma;

  if (n) {
    data->name = n;
    data->n++;
    data->sum += d;
    data->sumquad += d * d;
    mean = data->sum / data->n;
    sigma = sqrt ((data->sumquad - ((data->sum * data->sum) / data->n)) /
      (data->n));
    if (d < data->min) {
      data->min = d;
    }
    if (d > data->max) {
      data->max = d;
    }
  }
  else {
    mean = data->sum / data->n;
    sigma = sqrt ((data->sumquad - ((data->sum * data->sum) / data->n)) /
      (data->n));
  }
  if (print) {
    printf ("%s min=%.9f, max=%.9f, mean=%.9f, sigma=%.9f\n",
      n ? n : data->name, data->min, data->max, mean, sigma);
    if (n == NULL) {
      data->min = 1e37;
      data->max = 0.0;
    }
  }
}
static char pool[POOL_SIZE];
int
main (void)
{
  int i, free_mem;
  int j;
  size_t t;
  int n = 0;
  double s, first, last_h;
  void **m;
  minmax maldata, raldata, freedata;
  free_mem = init_memory_pool (POOL_SIZE, pool);
  printf ("Total free memory = %d\n", free_mem);
  init (&maldata);
  init (&raldata);
  init (&freedata);
```

```
 m = (void **) tlsf_malloc (NUM_MALLOC * sizeof (void *));
  for (i = 0; i < NUM_MALLOC; i++) {
    m[i] = NULL;
  }
  first = last_h = getcurtime ();
  for (i = 0; i < NUM_MALLOC; i++) {
    t = (size_t) (1 + drand48 () * SIZE_MALLOC);
    m[i] = tlsf_calloc (t, 1);
    if (((unsigned long) m[i] & (sizeof(void *) * 2 - 1)) != 0) {
      fprintf(stderr,"Alignment error %p\n", m[i]);
    }
#if FILL_MEM
    memset (m[i], -1, t);
#endif
  }
  for (j = 0; j < 1000; j++) {
    for (i = 0; i < NUM_MALLOC; i++) {
      if (m[i]) {
  t = (size_t) (1 + drand48 () * SIZE_MALLOC);
        s = getcurtime ();
  m[i] = tlsf_realloc (m[i], t);
        update (0, "realloc", s, getcurtime (), &raldata);
  if (((unsigned long) m[i] & (sizeof(void *) * 2 - 1)) != 0) {
    fprintf(stderr,"Alignment error %p\n", m[i]);
  }
#if FILL_MEM
        memset (m[i], -1, t);
#endif
      }
      if (m[i]) {
  s = getcurtime ();
  tlsf_free (m[i]);
  update (0, "free   ", s, getcurtime (), &freedata);
      }
      t = (size_t) (1 + drand48 () * SIZE_MALLOC);
      s = getcurtime ();
      m[i] = tlsf_malloc (t);
      update (0, "malloc ", s, getcurtime (), &maldata);
      if (((unsigned long) m[i] & (sizeof(void *) * 2 - 1)) != 0) {
        fprintf(stderr,"Alignment error %p\n", m[i]);
      }
#if FILL_MEM
      memset (m[i], -1, t);
#endif
    }
    n++;
    s = getcurtime ();
    if ((s - last_h) > 10) {
```

```c
      last_h = s;
      printf ("Count = %d %f\n",
        n * NUM_MALLOC, last_h - first);
      update (1, NULL, 0.0, getcurtime (), &maldata);
      update (1, NULL, 0.0, getcurtime (), &raldata);
      update (1, NULL, 0.0, getcurtime (), &freedata);
    }
  }
  for (i = 0; i < NUM_MALLOC; i++) {
    tlsf_free (m[i]);
  }
  tlsf_free (m);
  return 0;
}
```

Output:



```
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ cd /mnt/e/TLSF/src
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/src$ make
gcc -g -O2 -I -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-long -Wst
rict-aliasing=2  -DTLSF_USE_LOCKS=1 -DUSE_MMAP=1 -DUSE_SBRK=1   -c -o tlsf.o tlsf.c
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/src$ cd /mnt/e/TLSF/examples
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ make clean
rm -f -rf *.o test test?  *~ *.c.gcov *.gcda *.gcno
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ make
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2   -c -o test.o test.c
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2 -o test test.o ../src/tlsf.o
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2   -c -o test1.o test1.c
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2 -o test1 test1.o ../src/tlsf.o -lm
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2   -c -o test2.o test2.c
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2 -o test2 test2.o ../src/tlsf.o -lm
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2   -c -o test3.o test3.c
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2 -o test3 test3.o ../src/tlsf.o -lm
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2   -c -o test4.o test4.c
gcc -g -O2 -I../src -Wextra -Wall -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wno-long-lon
g -Wstrict-aliasing=2 -o test4 test4.o ../src/tlsf.o -lm
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ ./test
Total free memory= 1042208
Test OK
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ ./test1
Total free memory = 99993632
Count = 5870000 10.012442
malloc  min=0.000000000, max=0.000133038, mean=0.000000252, sigma=0.000000607
realloc min=0.000000000, max=0.000177145, mean=0.000000515, sigma=0.000000873
free    min=0.000000000, max=0.000164986, mean=0.000000240, sigma=0.000000588
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ ./test2
Count = 5760000 10.003246, max memory 50165216
malloc  min=0.000000000, max=0.000113010, mean=0.000000302, sigma=0.000000639
realloc min=0.000000000, max=0.000092030, mean=0.000000512, sigma=0.000000789
free    min=0.000000000, max=0.000073910, mean=0.000000235, sigma=0.000000584
Total used memory = 244896, max memory 51688560
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ ./test3
Pools' addresses: (0) 0x7f2e405fe040 (1) 0x7f2e40603040 (2) 0x7f2e40600840 (3) 0x7f2e40605840
Test OK
adi4608@LAPTOP-1UJU0JR4:/mnt/e/TLSF/examples$ ./test4
```

7. Results and Analysis

HERE ARE SOME COMPARISION GRAPHS AND TABLES OF DIFFERENT MEMORY MANAGEMENT ALGORITHMS according to the research papers and surveys.

It is possible to obtain the time complexity for the malloc and free operations from the TLSF pseudocode.

The malloc pseudo-code is the following:

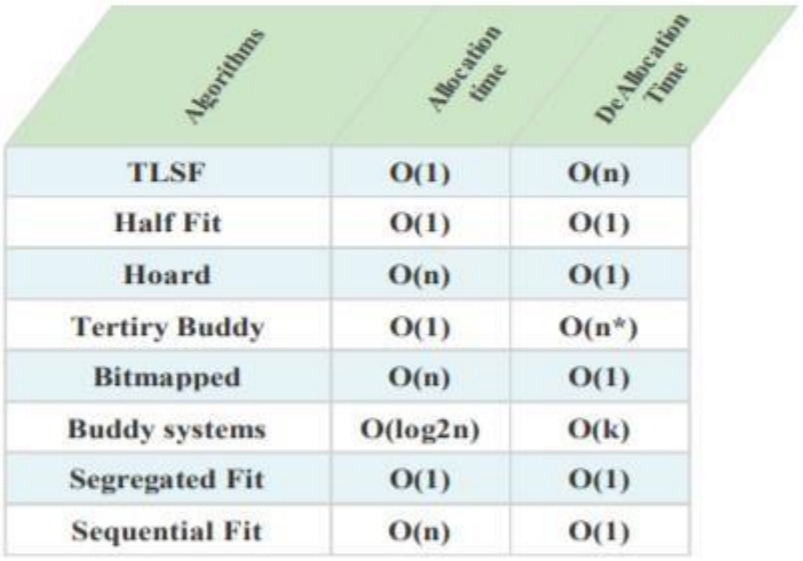
```
void *malloc(size){
  int fl, sl, fl2, sl2;
  void *found_block, *remaining_block;
  mapping (size, &fl, &sl);                        // O(1)
  found_block=search_suitable_block(size,fl,sl);// O(1)
  remove (found_block);                            // O(1)
  if (sizeof(found_block)>size) {
    remaining_block = split (found_block, size);
    mapping (sizeof(remaining_block),&fl2,&sl2);
    insert (remaining_block, fl2, sl2);            // O(1)
  }
  remove (found_block);                            // O(1)
  return found_block;
}
```

The pseudo-code of the free function is shown below:

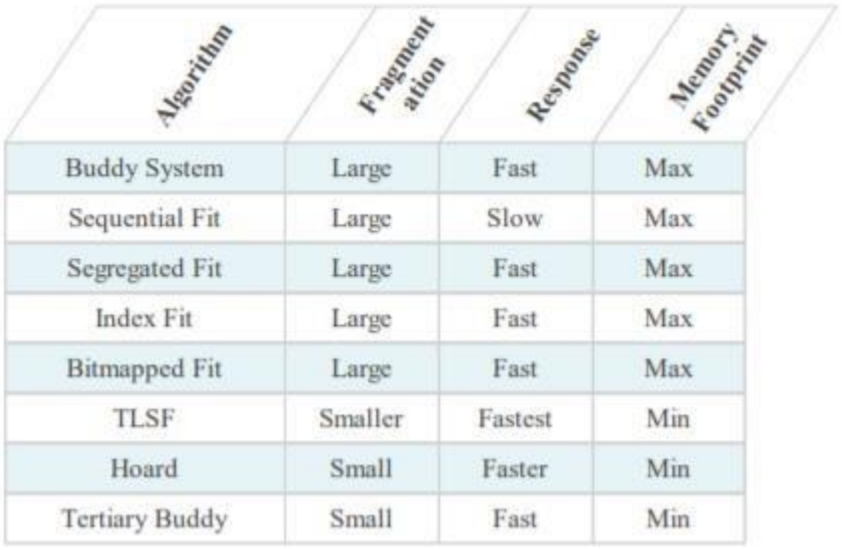
```
void free(block){
  int fl, sl;
  void *big_free_block;
  big_free_block = merge(block);                  // O(1)
  mapping (sizeof(big_free_block), &fl, &sl);
  insert (big_free_block, fl, sl);                // O(1)
}
```

The asymptotic worst case response time is:

malloc()	free()
O(1)	O(1)



Algorithms	Allocation time	DeAllocation Time
TLSF	O(1)	O(n)
Half Fit	O(1)	O(1)
Hoard	O(n)	O(1)
Tertiry Buddy	O(1)	O(n*)
Bitmapped	O(n)	O(1)
Buddy systems	O(log2n)	O(k)
Segregated Fit	O(1)	O(1)
Sequential Fit	O(n)	O(1)



Algorithm	Fragment ation	Response	Memory Footprint
Buddy System	Large	Fast	Max
Sequential Fit	Large	Slow	Max
Segregated Fit	Large	Fast	Max
Index Fit	Large	Fast	Max
Bitmapped Fit	Large	Fast	Max
TLSF	Smaller	Fastest	Min
Hoard	Small	Faster	Min
Tertiary Buddy	Small	Fast	Min

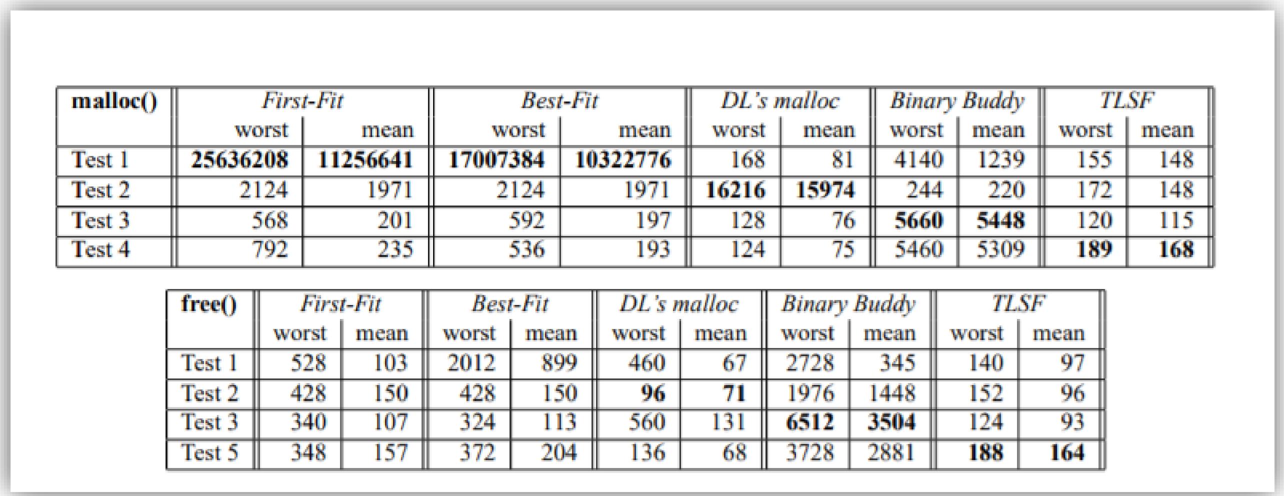
The workload that drives a simulation or a final implementation can be obtained in two ways: –

- Using real workloads, which allow to evaluate the algorithm in real situations. Most of the research results on dynamic allocation have been obtained by using well known programs such as compilers (gcc, perl, etc.) or application programs (cfrac, espresso, etc.) as real workloads due to their intensive use of dynamic memory. *However, real-time requirements have not been considered because of a lack of examples of use (the use of dynamic memory has usually been considered inappropriate for this kind of applications).*

- Using synthetic workloads by extracting and modelling the events from a program at run time or by generating events randomly based on a probabilistic model.

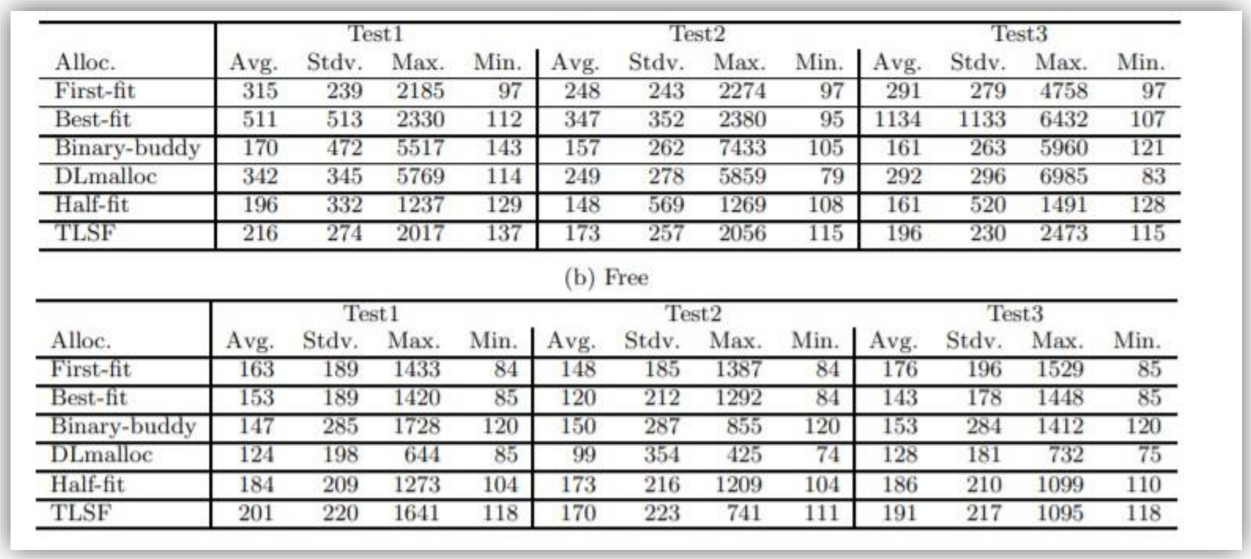
Therefore, to obtain experimental results that really show the performance parameters relevant to real-time applications, synthetic workload has been used. Each test workload has been designed trying to produce the worst-case scenario for each allocator. The set of allocators analyzed were:

First-Fit, Best-Fit, Douglas Lea's malloc, Binary Buddy, and TLSF based on papers and survey.



malloc()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	worst	mean	worst	mean	worst	mean	worst	mean	worst	mean
Test 1	**25636208**	**11256641**	**17007384**	**10322776**	168	81	4140	1239	155	148
Test 2	2124	1971	2124	1971	**16216**	**15974**	244	220	172	148
Test 3	568	201	592	197	128	76	**5660**	**5448**	120	115
Test 4	792	235	536	193	124	75	5460	5309	**189**	**168**

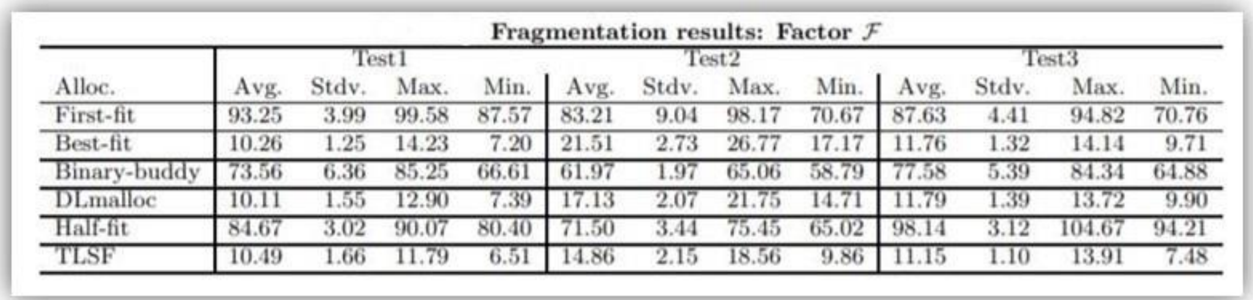
free()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	worst	mean	worst	mean	worst	mean	worst	mean	worst	mean
Test 1	528	103	2012	899	460	67	2728	345	140	97
Test 2	428	150	428	150	**96**	**71**	1976	1448	152	96
Test 3	340	107	324	113	560	131	**6512**	**3504**	124	93
Test 5	348	157	372	204	136	68	3728	2881	**188**	**164**



	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	315	239	2185	97	248	243	2274	97	291	279	4758	97
Best-fit	511	513	2330	112	347	352	2380	95	1134	1133	6432	107
Binary-buddy	170	472	5517	143	157	262	7433	105	161	263	5960	121
DLmalloc	342	345	5769	114	249	278	5859	79	292	296	6985	83
Half-fit	196	332	1237	129	148	569	1269	108	161	520	1491	128
TLSF	216	274	2017	137	173	257	2056	115	196	230	2473	115

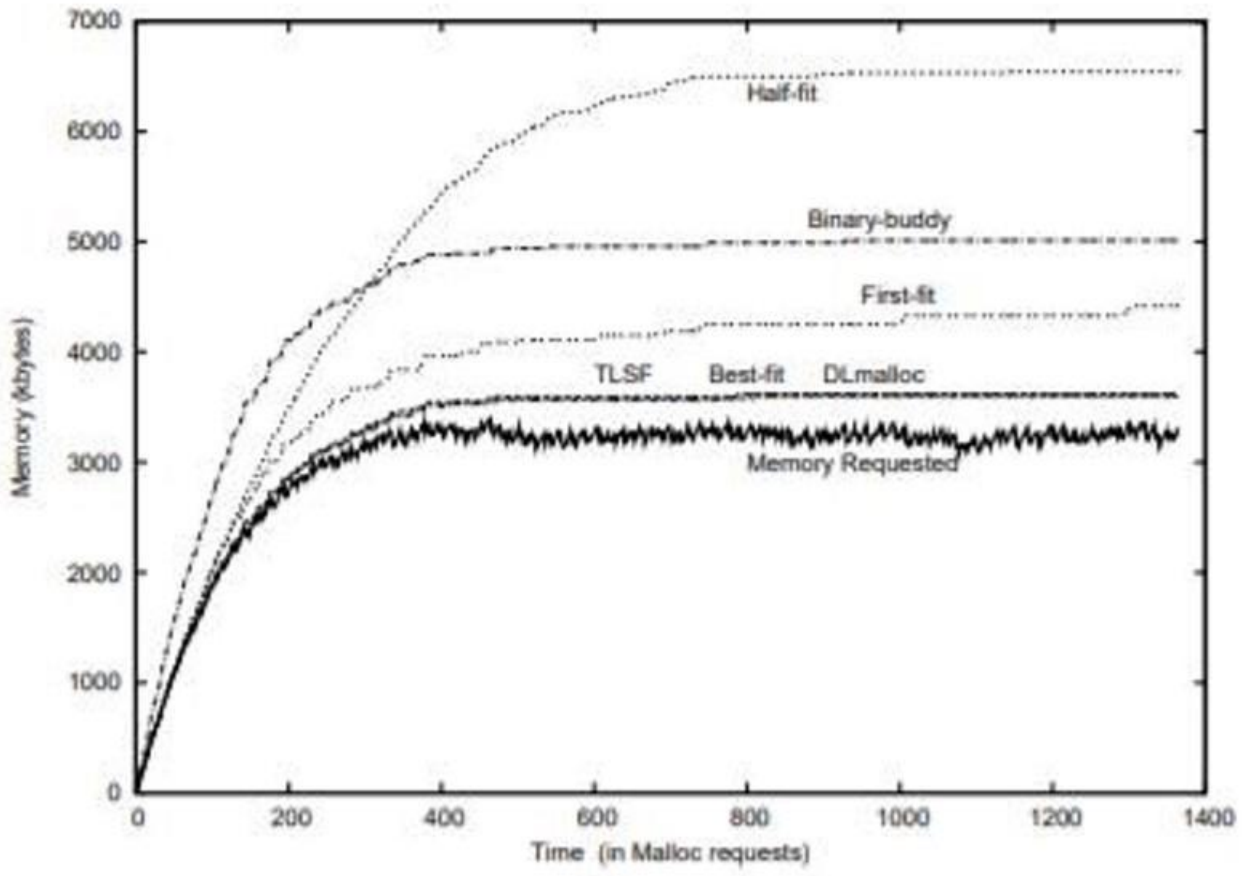
(b) Free

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	163	189	1433	84	148	185	1387	84	176	196	1529	85
Best-fit	153	189	1420	85	120	212	1292	84	143	178	1448	85
Binary-buddy	147	285	1728	120	150	287	855	120	153	284	1412	120
DLmalloc	124	198	644	85	99	354	425	74	128	181	732	75
Half-fit	184	209	1273	104	173	216	1209	104	186	210	1099	110
TLSF	201	220	1641	118	170	223	741	111	191	217	1095	118



Fragmentation results: Factor $\mathcal{F}$

	Test1				Test2				Test3			
Alloc.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.	Avg.	Stdv.	Max.	Min.
First-fit	93.25	3.99	99.58	87.57	83.21	9.04	98.17	70.67	87.63	4.41	94.82	70.76
Best-fit	10.26	1.25	14.23	7.20	21.51	2.73	26.77	17.17	11.76	1.32	14.14	9.71
Binary-buddy	73.56	6.36	85.25	66.61	61.97	1.97	65.06	58.79	77.58	5.39	84.34	64.88
DLmalloc	10.11	1.55	12.90	7.39	17.13	2.07	21.75	14.71	11.79	1.39	13.72	9.90
Half-fit	84.67	3.02	90.07	80.40	71.50	3.44	75.45	65.02	98.14	3.12	104.67	94.21
TLSF	10.49	1.66	11.79	6.51	14.86	2.15	18.56	9.86	11.15	1.10	13.91	7.48



8. Future Direction

When TLSF is compared with other well-known allocators, the results obtained can be considered as good as the best. Whereas all allocators present good results in response time, there is a significant difference in the total amount of memory needed to allocate a workload. Half-fit and binary-buddy fragmentation are not appropriate for embedded systems when memory is limited. On the other hand, DLmalloc and TLFS reach similar results in fragmentation. Considering both measurements (temporal and spatial), we can conclude that TLSF has the best performance of all compared allocators under the synthetic workload generated.

There are still some open issues, and in particular the determination of appropriated workloads for evaluation purposes. It is difficult to find examples of real-time systems using dynamic memory (because of the unpredictability of current allocators). Also, calculating the maximum live memory of a program is difficult, although it can be integrated with WCET analysis techniques which already perform dataflow analysis of programs.
Additional studies are required to investigate a wider variety of real-time applications, both to measure their performance on the real applications and to determine the degree to which arbitrary code complies with the stated requirements of this project. Also, the integration of this algorithm in a memory resource management able to provide quality of service considering the CPU and memory management in an integrated way is future work.

9. Conclusion

In the end we want to conclude that in the memory management of operating systems there are various types of memory allocation algorithms but some of the memory allocations are not suited in some special cases of OS's such as RTOS and Embedded systems.
The main problems of memory allocation algorithms, such as the lack of bounded and fast response time primitives, and the high fragmentation, have been a limitation for the use of dynamic memory in real-time and general-purpose embedded systems. Several DSA algorithms, as Buddy systems, have improved some aspects like the response time, but maintaining the fragmentation in non-acceptable limits. In this project, we have presented a new algorithm called TLSF, based on two levels of segregated lists which provides explicit allocation and deallocation of memory blocks with a temporal cost Θ(1). We also reviewed several works on TLSF and found out comparison between some already introduced algorithms and made a table based on experimental results.

9. References

[1] - Durgesh Raghuvanshi "Memory Management in Operating System" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-2 | Issue-5, August 2018, pp.23462347, URL: https://www.ijtsrd.com/papers/ijtsrd18342.pdf

[2] - Kumar, Dinesh & Singh, Mandeep. (2019). MEMORY MANAGEMENT IN OPERATING SYSTEM. 6. 465-471.

[3] - Abdullah Awais, M. (2016). Memory Management: Challenges and Techniques for traditional Memory

Allocation Algorithms in Relation with Today's Real Time Needs. Advances In Computer Science : An International Journal, 5(2), 22-27. Retrieved from http://www.acsij.org/acsij/article/view/459

[4] - IvyPanda. (2020, March 30). Computer's Memory Management. Retrieved from https://ivypanda.com/essays/computers-memory-management

[5] - Miguel Masmano, Ismael Ripoll, Patricia Balbastre & Alfons Crespo***. A constant-time dynamic storage allocator for real-time systems*** https://link.springer.com/article/10.1007/s11241-008-9052-7