Description of the TLSF memory allocator

P. Hamshere

1. INTRODUCTION

This paper describes the TLSF memory allocation approach.

All examples use an architecture with 32 bit words and a 32 bit memory address bus.

The algorithm is described in documents found here: http://www.gii.upv.es/tlsf/. Specifically these two articles: http://www.gii.upv.es/tlsf/files/jrts2008.pdf and http://www.gii.upv.es/tlsf/files/spe_2008.pdf.

Code examples are from Matt Conte's implementation as described here: https://github.com/mattconte/tlsf

2. TLSF SEGMENTATION ALGORITHM

2.1 Overview

A memory allocation algorithm must keep a record of free memory and provide a suitably sized allocation of free memory in response to a memory allocation requrest.

TLSF divides memory into blocks of pre-determined sizes and maintains a list of free blocks for each block size. TLSF has some optimisations for small block sizes. TLSF aligns all requuests to the nearest word.

The TLSF algorithm ensures a fixed-time look-up to retrieve a suitable sized block of memory. It does this using two levels of segmentation, each indexed by a word-sized bitmap. The determine the index into an array of block free list headers. The algorithm also rounds up the size request to the next block size and retrieves the first block in that list, which will always provide a block large enough to meet the request.

2.2 Segmentation and the index bitmaps

Memory is segmented in two levels. The first level has 32 segments and each segment spans 2^n bytes, where *n* represents the number of the level (0 to 31). For example, segment 10 would be of size 2^{10} = 1024 and would cover memory ranges 1024 to 2043. Each higher segment is double the size of the next lower segment.

The allocation sizes for lower indices (0 to 6) are small and it is therefore not efficient to use these. This is covered later in the small block optimisation description.

The second level then splits each of these segments into 32 blocks of the same size. For segment 10, this would split the segment of 1024 into 32 blocks of 32 bytes each. The first block covers 1024 to 1057, the second block 1058 to 1089 and so on.

Bitmaps are kept to show whether there are any free blocks of a particular size. The free block list headers are stored in an array with an element for each block size.

The fist level bitmap is *fl_bitmap* and has a bit set if there is any free block in that segment range. The index into this bitmap is *fl* which can have values from 0 to 31.

The second level bitmap is *sl_bitmap*, which is an array of bitmaps with an entry for each value of *fl.* This array has a bit set if there are any free blocks that map to that segment size. The index into this bitmap is *sl*, which can also have values from 0 to 31.
Figure 1 shows an example layout for *fl* ranging from 7 to 10.

sl

fl		0	1	2	3	4		27	28	29	30	31
	7	128-131	132-135	136-139	140-143	144-147		236-239	240-243	244-247	248-251	252-255
	8	256-263	264-271	272-279	280-287	288-295		472-479	480-487	488-495	496-503	504-511
	9	512-527	528-543	544-559	560-575	576-591		944-959	960-975	976-991	992-1007	1008-1023
	10	1024-1055	1056-1087	1088-1119	1120-1151	1152-1183		1888-1919	1920-1951	1952-1983	1984-2015	2016-2047

Figure 1 – *fl* layout

To locate a block of *size = 530* (range 528 – 543) would result in *fl = 9* and *sl = 1*.
To locate a block of *size = 258* (range 256 - 263) would result in *fl = 8* and *sl = 0.*

Figure 2 shows an example of the *fl_bitmap* and *sl_bitmap* contents. Each bit set in *fl_bitmap* indicates that the corresponding row in *sl_bitmap* has some free blocks. Each bit set in *sl_bitmap[fl]* indicates the the corresponding free list has at least one free block on it.

An example of *fl = 8* (regardless of *sl*) would indicate no free blocks in that size range as that bit in *fl_bitmap* is not set.
An example of *fl =9* shows there are free blocks in the row *sl_bitmap[9].* Checking for *sl = 1* shows there are free blocks in that list.

fl_bitmap

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																			1	1	1	1	0	1							

sl_bitmap[FL_INDEX_COUNT]

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7	0	1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	1	1	1	0	0	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	1	0
10	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	0
11	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
12	0	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0	1	0	1	0	1	1	0	1	1	0	1	1	0	0	0	0

Figure 2 – *fl_bitmap* and *sl_bitmap*

The indexes *fl* and *sl* can be easily obtained from the size of memory requested.
As the block sizes increase in powers of 2, the *fl* index relates to the bit position of the highest bit in the size requested.
The *sl* index is then the next five bits in the size requested.
Figure 3 illustrates this.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	0	0	1	1	1	0	1

$fl = 9 \qquad sl = 25$

Figure 3 – bit locations of *fl* and *sl*

The last four bits are not needed for the block segment – they create the lower and upper bound of the range for that block.

2.3 The free block list

The heads of the lists of free blocks are maintained in the *blocks* array. This has an entry for each bit in *sl_bitmaps*.

Figure 4 shows the array of free lists. For illustration purposes the head of a free block list is indicated by a grey cell. (Note this has a reversed ordering from the bitmap above but represents the same data).

blocks[FL_INDEX_COUNT][SL_INDEX_COUNT]

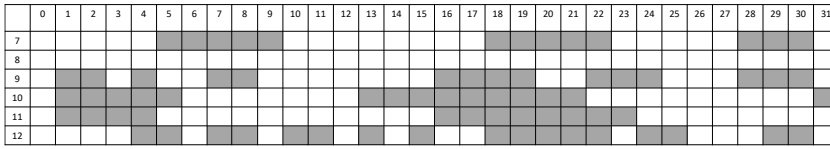

Figure 4 – free lists

Once the head of the list is located it is possible to add, remove or merge blocks on the list (or in physical memory).

2.4 Small block size optimisation

As the second level segment creates 32 ranges and the word size is 4 bytes then the minimum meaningful size to subdivide is 128, giving allocation of memory sizes in the range 0 to 127 in steps of 4 bytes.

The small block optimisation uses the *fl=0* level to represent these blocks from 0 to 127 in size. As the normal setting for *fl* for blocks of size 128 or more is *fl=7* there is now a gap in the table from *fl=0* to *fl=7*. To resolve this the *fl* index is reduced by 6 to map into the *sl_bitmap* array.

For example, for the *fl=9, sl=1* example above, *fl* would be reduced to *fl=3* to map to *sl_bitmap[3]*.

Figure 5 shows the new lower size ranges, with both the original and amended *fl* value shown on the left.

		sl									
		0	1	2	3		27	28	29	30	31
0	0	0-3	4-7	8-11	12-15		108-111	112-115	116-119	120-123	124-127
7	1	128-131	132-135	136-139	140-143		236-239	240-243	244-247	248-251	252-255
8	2	256-263	264-271	272-279	280-287		472-479	480-487	488-495	496-503	504-511
9	3	512-527	528-543	544-559	560-575		944-959	960-975	976-991	992-1007	1008-1023
10	4	1024-1055	1056-1087	1088-1119	1120-1151		1888-1919	1920-1951	1952-1983	1984-2015	2016-2047

fl

Figure 5 – small block size optimisation

2.5 Deriving the fl and sl index

The first level segregation range aligns to the top set bit of the size request *size*, followed by all zeros or all ones. For example, if bit 7 is set (which is decimal 128) then this maps the range 128 to 255.

The *fl* index is therefore the number of the highest bit set in the size request *size*.

The *sl* index is represented by the next 5 bits in the size request.

Figure 5 illustrates this.

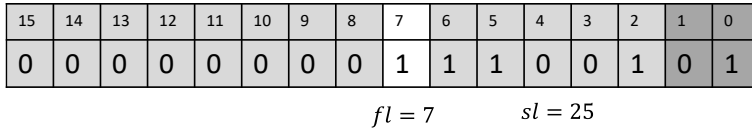

Figure 6 – locating *fl* and *sl*

2.6 Alignment and request rounding to next block size

Any request for a memory region of *size* is aligned to the nearest word.

To make the free list retrieval have a fixed execution duation there is no searching of the list for a best-sized block. TLSF instead rounds the *size* request up to the next block size, so that the fist block in the free list will meet the need of the request.

3. DESCRIPTION OF THE CODE

3.1 Main functions to find a free block

This section describes how a suitably sized block greater than *size* is found and returned to the caller.

The standard memory allocation is *malloc()* and the TLSF implementation is *tlsf_malloc()*.
To make the code slightly more readable all casts have been removed.

The *tlsf_malloc()* function first determines the adjustment to align to a word boundary. It then uses the *block_locate_free()* function to find the free block, then processes it as used (removing frrom the free list) in *block_prepare_used()*.

tlsf_malloc()

```
void* tlsf_malloc(tlsf_t tlsf, size_t size)
{
    control_t* control = tlsf;
    const size_t adjust = adjust_request_size(size, ALIGN_SIZE);
    block_header_t* block = block_locate_free(control, adjust);
    return block_prepare_used(control, block, adjust);
}
```

Adjusting the requested size is performed in *adjust_request_size()* which rounds the requested size to the next word boundary.

For example, if a size of 51 is requested this will return 52 which is aligned to a word boundary.

adjust_request_size ()

```
static size_t adjust_request_size(size_t size, size_t align)
{
    size_t adjust = 0;
    if (size)
    {
        const size_t aligned = align_up(size, align);
        if (aligned < block_size_max)
        {
            adjust = tlsf_max(aligned, block_size_min);
        }
    }
    return adjust;
}
```

Finding a free block is done by *block_locate_free()*. This takes the adjusted *size* and uses *mapping_search()* to find the *fl* and *sl* indexes for the block range that covers that *size*. This does not find a free block but only the two indices to the block with a range covering *size*.
Once that base location is determined, *search_suitable_block()* is called to find the actual next free block.
After that, it is removed from the free list and returned.

block_locate_free ()

```
static block_header_t* block_locate_free(control_t* control, size_t size)
{
   int fl = 0, sl = 0;
   block_header_t* block = 0;

   if (size)
   {
      mapping_search(size, &fl, &sl);

      if (fl < FL_INDEX_COUNT)
      {
        block = search_suitable_block(control, &fl, &sl);
      }
   }

   if (block)
   {
      remove_free_block(control, block, fl, sl);
   }

   return block;
}
```

The *mapping_search()* function first rounds the block up to the next block size. This is done to avoid having to search a free block list for a block large enough for the request – the first free block in the next sized free block list will always meet the need.

After rounding up it calls *mapping_insert()* to obtain the correct *fl* and *sl* indices for that block size.

mapping_search ()

```
static void mapping_search(size_t size, int* fli, int* sli)
{
    if (size >= SMALL_BLOCK_SIZE)
    {
        const size_t round = (1 << (tlsf_fls_sizet(size) -
 SL_INDEX_COUNT_LOG2)) - 1;
        size += round;
    }
     mapping_insert(size, fli, sli);
}
```

The *mapping_insert()* function which finds the *fl* and *sl* indices for that block size.
If *size* falls into the small block optimisation size range, then *fl=0* and *sl* will be a simple division of *size* by the size of each small block (4 bytes).
If *size* is larger than that then *fl* and *sl* are calculated. *fl* is obtained by locating the highest set bit in size and using the number of the bit location, adjusted to remove the offset caused by the small block optimisation.
sl is then the following 5 bits.
The details of these calculations are described in the next section.

mapping_insert ()

```
static void mapping_insert(size_t size, int* fli, int* sli)
{
   int fl, sl;
   if (size < SMALL_BLOCK_SIZE)
   {
     /* Store small blocks in first list. */
     fl = 0;
     sl = size / (SMALL_BLOCK_SIZE / SL_INDEX_COUNT);
   }
   else
   {
      fl = tlsf_fls_sizet(size);
      sl = size >> (fl - SL_INDEX_COUNT_LOG2)) ^ (1 <<
SL_INDEX_COUNT_LOG2);
      fl -= (FL_INDEX_SHIFT - 1);
   }
   *fli = fl;
   *sli = sl;
}
```

Once the indices are obtained, the free list can be checked to see if there are free blocks.

The function first checks for a free block for the indicated block size and higher sized blocks on the current *sl_bitmap*.

If there are none, then the *fl_bitmap* is checked to see if there are any higher values of *fl* with free blocks. If there are then another value of *fl* is used and another value of *sl* determined, else an error is returned to show there is no available block.

search_suitable_block ()

```
static block_header_t* search_suitable_block(control_t* control, int* fli, int*
sli)
{
    int fl = *fli;
     int sl = *sli;

    /*
    ** First, search for a block in the list associated with the given
    ** fl/sl index.
    */
    unsigned int sl_map = control->sl_bitmap[fl] & (~0U << sl);
    if (!sl_map)
    {
        /* No block exists. Search in the next largest first-level list. */
        const unsigned int fl_map = control->fl_bitmap & (~0U << (fl + 1));
        if (!fl_map)
         {
            //* No free blocks available, memory has been exhausted. */
            return 0;
         }
        fl = tlsf_ffs(fl_map);
       *fli = fl;
        sl_map = control->sl_bitmap[fl];
    }
    tlsf_assert(sl_map && "internal error - second level bitmap is null");
   sl = tlsf_ffs(sl_map);
   *sli = sl;

    /* Return the first block in the free list. */
    return control->blocks[fl][sl];
}
```

3.2 Description of paramenters used

These are the key parameters used to set the sizes of the bitmaps and arrays.

Parameter	Description	Calculation
SL_INDEX_COUNT_LOG2	Number of bits in the SL index	5 (32 bits)
ALIGN_SIZE_LOG2	Number of bytes to align	2 (4 bytes)
ALIGN_SIZE	Number of bytes to align	1<< ALIGN_SIZE_LOG2 4
FL_INDEX_MAX	Largest value of *fl* required to span memory	19
SL_INDEX_COUNT	Largest value of *sl*	1 << SL_INDEX_COUNT_LOG2 32
FL_INDEX_SHIFT	Number of bits for *sl_bitmap* plus the alignment bits	SL_INDEX_COUNT_LOG2 + ALIGN_SIZE_LOG2 7
FL_INDEX_COUNT	Revised largest value of *fl* to account for small block optimisaion	FL_INDEX_MAX - FL_INDEX_SHIFT + 1 13
SMALL_BLOCK_SIZE	Top of the small block range	1 << FL_INDEX_SHIFT 128

3.3 Explanation of bit manipulation functions

Two bit search functions are used in the algorithm.

ffs finds the first bit set in the word, returning the index of that bit (0-31). FFS stands for Find First Set.

fls finds the last bit set in the word, returning the index of that bit (0-31). FLS stands for Find Last Set.

Many processors have built in equivalents to *ffs* and *fls*, or alternative instructions that can be used to optimise execution time. Examples are *clz, CountLeadingZeros, BitScanForward, BitScanReverse.*

The code has a generic algorithm to use where there are no processor-specific instructions available.

An example that uses *clz* will count the leading zeros, then subtract from 32 and then subtract 1. If the highest set bit is in bit 30 then there is one leading zero in bit 31, so this would return 32 -1 – 1 = 30.

tlsf_fls ()

```
tlsf_decl int tlsf_fls(unsigned int word)
{
   const int bit = word ? 32 - __clz(word) : 0;
   return bit - 1;
}
```

The generic algorithm is design to have no loops which would cause a variation in timing.

tlsf_fls_generic ()

```
tlsf_decl int tlsf_fls_generic(unsigned int word)
{
   int bit = 32;

   if (!word) bit -= 1;
   if (!(word & 0xffff0000)) { word <<= 16; bit -= 16; }
   if (!(word & 0xff000000)) { word <<= 8; bit -= 8; }
   if (!(word & 0xf0000000)) { word <<= 4; bit -= 4; }
   if (!(word & 0xc0000000)) { word <<= 2; bit -= 2; }
   if (!(word & 0x80000000)) { word <<= 1; bit -= 1; }
   return bit;
}
```

The calculation for *ffs* uess *fls*.

tlsf_ffs_generic ()

```
tlsf_decl int tlsf_ffs(unsigned int word)
{
   return tlsf_fls_generic(word & (~word + 1)) - 1;
}
```

This works by isolating the first set bit and using *fls* to locate that. Figure 7 shows an example of this.

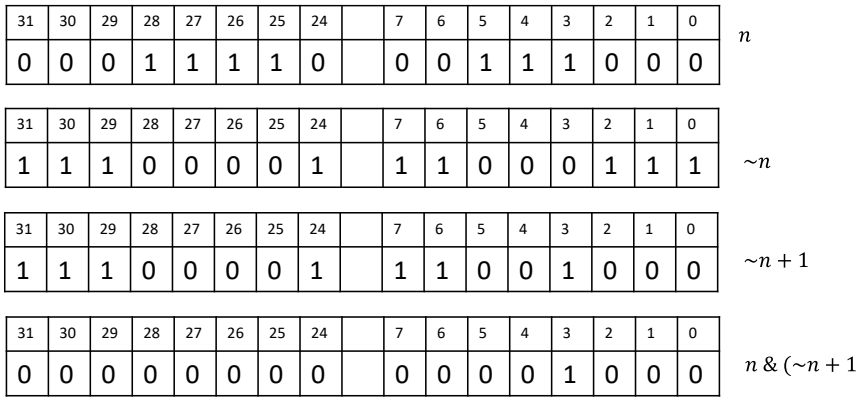

Figure 7 – example of *fls*

Two other functions use interesting bit manipulation – *align_up* and *align_down.*

align_up ()

```
static size_t align_up(size_t x, size_t align)
{
   return (x + (align - 1)) & ~(align - 1);
}
```

Figure 8 shows a worked example of *align_up*.

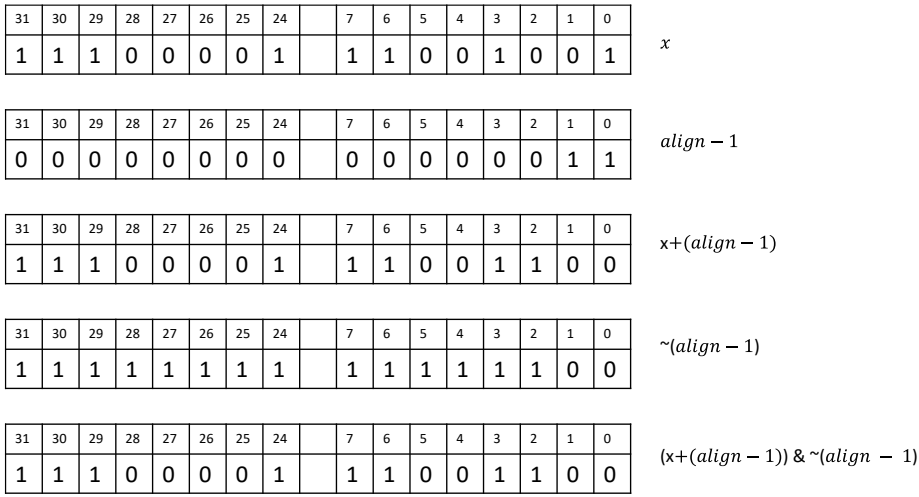

Figure 8 – example of *align_up*

align_down ()

```
static size_t align_down(size_t x, size_t align)
{
    return x - (x & (align - 1));
}
```

Figure 9 shows a worked example of *align_down*.

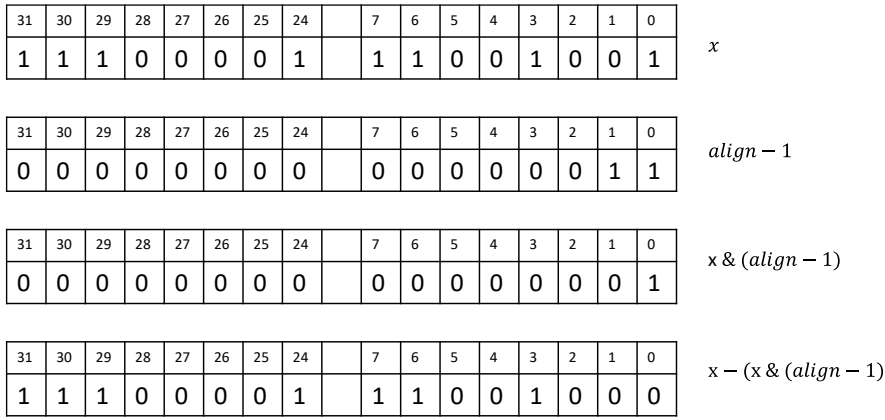

Figure 9 – example of *align_down*

4. FREE LIST MANAGEMENT AND DATA STRUCTURES

4.1 Memory pools

The memory allocated to TLSF to control is called a memory pool. It can manage multiple memory pools and for the purposes of TLSF they are just added into the list of free blocks.

Each pool, therefore, starts as one large block of the size of the pool (less overhead) and is added to the free list for that size. As it gets used, the pool (the one large block) gets divided into smaller blocks, just like any other block. In this way, once a pool (large block) is added to the free list, it is no different from any other block.

Figure 10 shows three empty pools. Each pool is a single large bock and they will be added into the free block list in *control_t.*

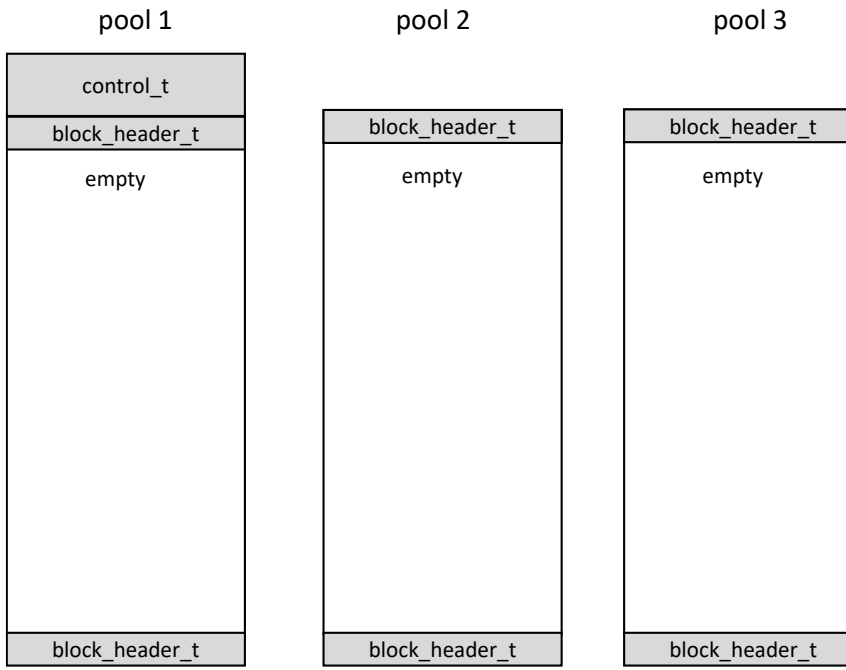

Figure 10 – empty pools

Once blocks are allocated from the pools, they will look more Figure 11. Each empty block will be on the approproate free block list.

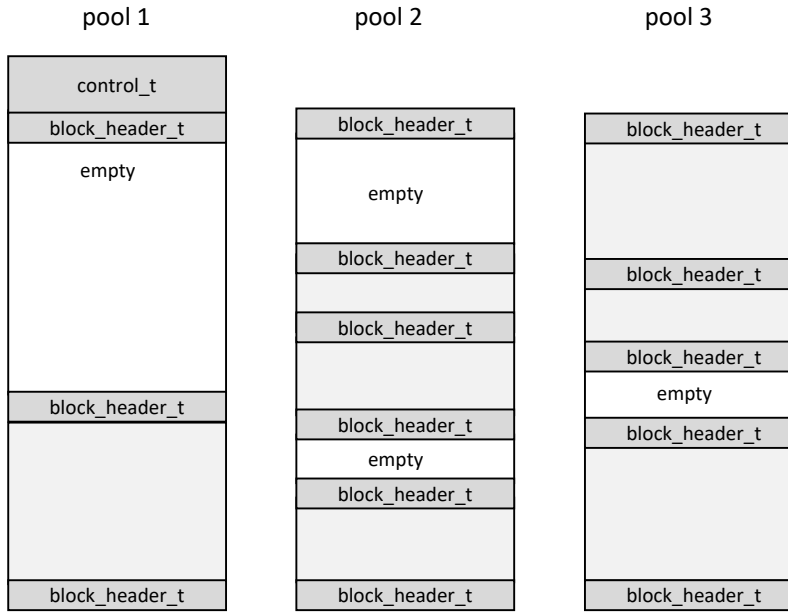

Figure 11 – pools with allocated or free blocks

4.2 Free list management

Free list management is key aspect of ensuring the algorithm has a bounded execution time.

We have already seen that the head of each free list queue is held in the *blocks* array. This is stored in a structure called *control_t*, along with the *fl_bitmap* and *sl_bitmap*.

control_t needs to be stored in allocated memory, usually at the beginning of the *memory pool*.

struct control_t

```
typedef struct control_t
{
    /* Empty lists point at this block to indicate they are free. */
    block_header_t block_null;

    /* Bitmaps for free lists. */
    unsigned int fl_bitmap;
    unsigned int sl_bitmap[FL_INDEX_COUNT];

    /* Head of free lists. */
    block_header_t* blocks[FL_INDEX_COUNT][SL_INDEX_COUNT];
} control_t;
```

Each block of free or allocated memory starts with the header *block-_header_t.* This has the purpose of storing the size of the block, and if the block is free it also stores the free list pointers and a pointer to the start of the prior block.

Figure 12 shows a view of the *control_t* structure.

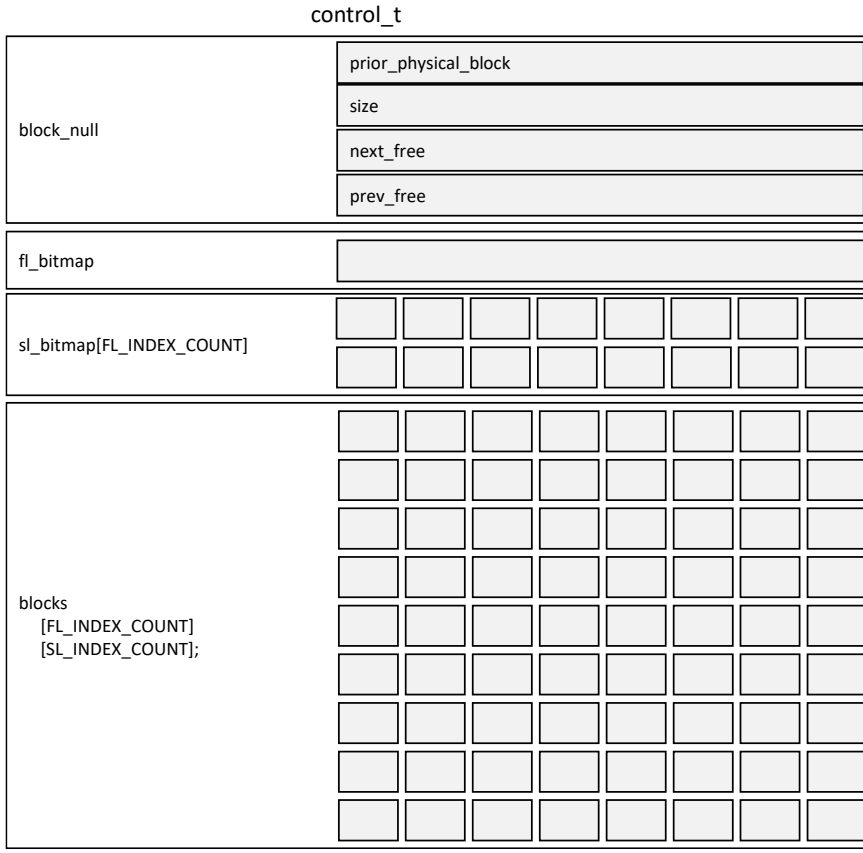

Figure 12 – *control_t*

Figure 13 shows a TLSF *memory pool* with only the control_t structure, with the *block_header_t* for the main block and a sentinel 0 sized end block, and also with multiple *block_header_t* headers for the memory blocks.

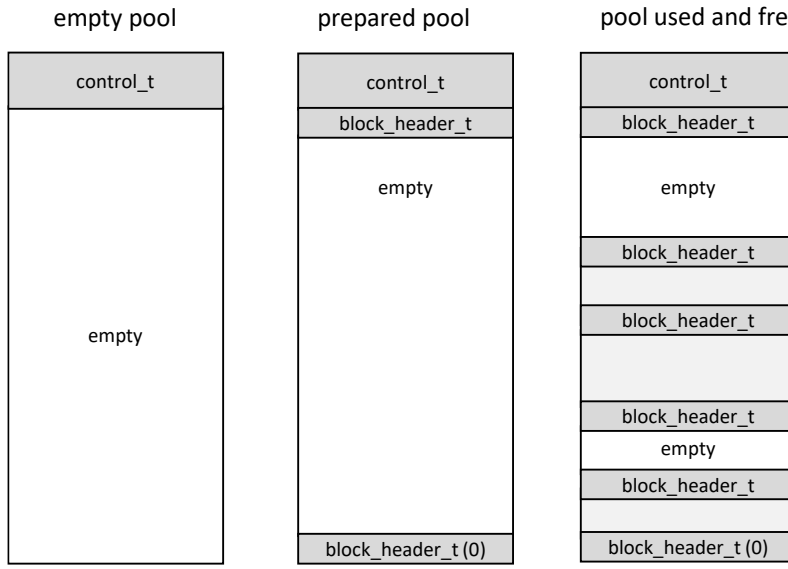

Figure 13 – prepared pool

Each block of free or allocated memory starts with the header *block_header_t.* This has the purpose of storing the size of the block, and if the block is free it also stores the free list pointers and a pointer to the start of the prior block.

Figure 14 shows blocks linked into the free lists pointed to in *control_t.*

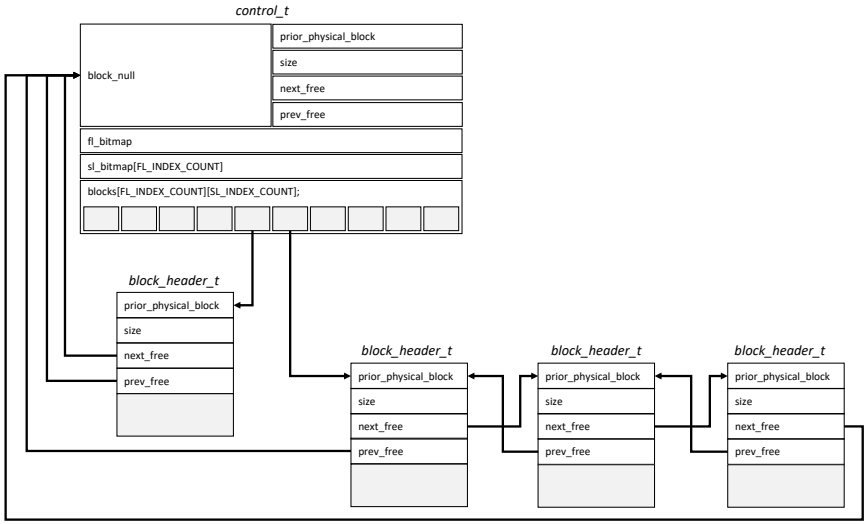

Figure 14 – free blocks on the linked list

block_header_t

```
typedef struct block_header_t
{
    /* Points to the previous physical block. */
    struct block_header_t* prev_phys_block;

    /* The size of this block, excluding the block header. */
    size_t size;

    /* Next and previous free blocks. */
    struct block_header_t* next_free;
    struct block_header_t* prev_free;
} block_header_t;
```

The block header *block_header_t* has a structure as shown in Figure 15.

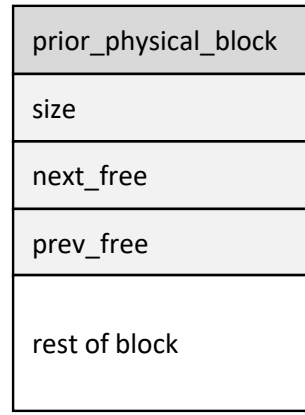

Figure 15

If the block is in use, *block_header_t* holds the size of the block. In this case *next_free* and *prev_free* are ignored, and their space used by the data in the block.

If the block is free, then *block_header_t* holds the size and pointers to the previous and next free blocks in *next_free* and *prev_free*.

The size of the block will always be a multiple of the size of a word. This means that the lower two bits of *size* are free to be used as flags.

Bit 0 is used to show whether the block is free or in use. Bit 1 is used to show whether the previous block is free or in use.The bit is set to show free, and cleared to show in use. Figure 16 illustrates this.

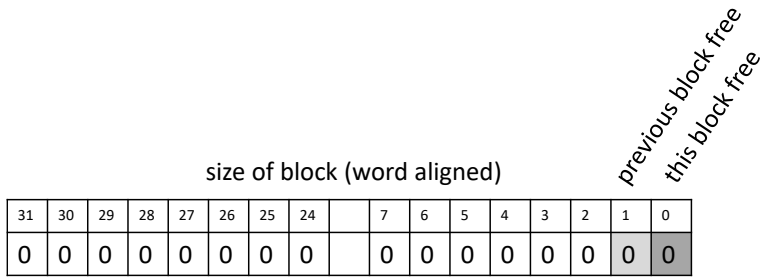

Figure 16

Flagging whether the previous block is free is important because for a free block the final word in the block is a pointer to the start of that block, and that field only has meaning if the block is free. This allows *tlsf_free* to check the following block and the prior block to see if either or both can be merged with the block being freed.

The first word in the structure actually belongs to the previous block. If that block is used then this this is simply the last word of the prior block. If the prior block is empty then this word is a pointer to the start of the block before that – the prior physical block. This allows the start of the previous block to be located and merged with the current block in the event that both are empty.

Figure 17 shows the four options where the current block and prior block are either free or in use, to indicate there the *prior_physical_block* field is used.

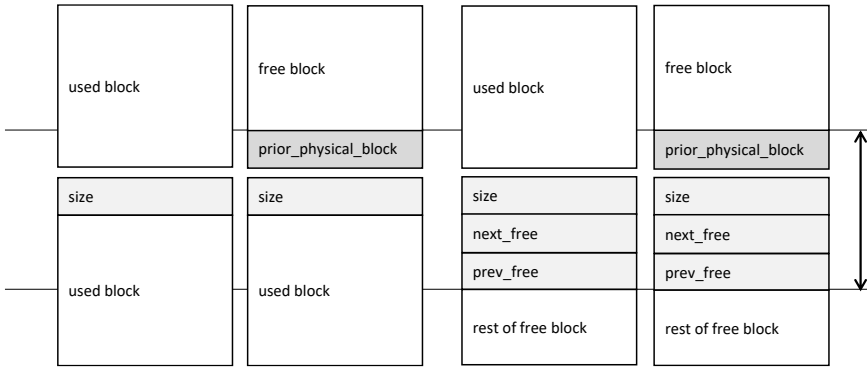

Figure 17 – prior block free / used and current block free / used

Figure 18 shows the physical memory layout for a free block, a used block and then a free block. The first column shows the actual memory and the other two columns show the in-use and free respresentations in full.

Because of this use of *prior_physical_block* the pointer to a block, which is a pointer to the start of *block_header_t*, points to one word before the size field of the block and two words before the actual data in the block. It is actually pointing to the last word of the previous block. There are functions to move between this pointer to *block_header_t* and the start of the block data.

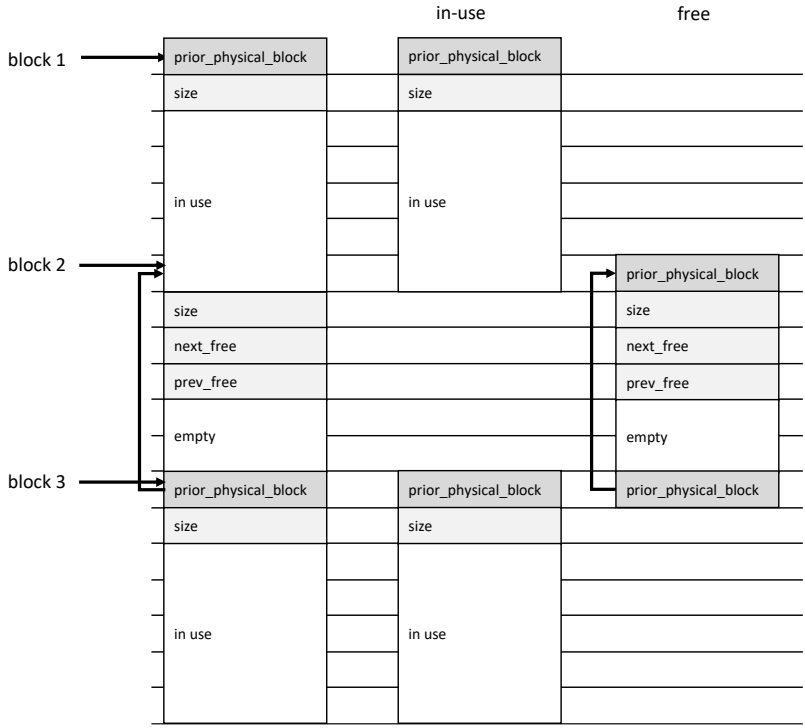

Figure 18 – physical memory layout

5. BLOCK INTEGRITY CHECKING

5.1 Poisoning

In the ESP32 implementation there is the opportunity to ‘poison’ each used block with a header and footer which facilitates integrity checking.

The ‘poisoned’ block contents now looks like Figure 19. The header and footer are called ‘canaries’ and contain fixed, known data. This can be checked, and if at any point the ‘canaries’ are corrupted then the block will have lost its integrity.

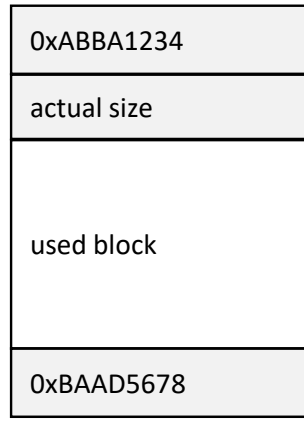

Figure 19

Figure 20 shows how the new ‘poisoned’ block sits within the block structures.

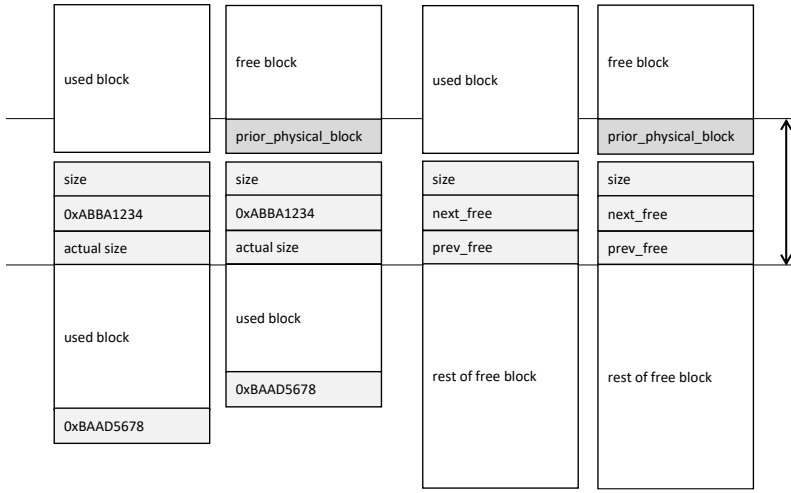

Figure 20

The header and actual size fields occupy the same space as the *next_free* and *prev_free* fields in a free block.

The function calls for the poisoned allocations wrap the existing TLSF calls and move the pointers accordingly – so the pointer to the actual data is move to the TLSF view before calling the TLSF functions.