Description of the TLSF Memory Allocator Version 2.0

M. Masmano, I. Ripoll, A. Crespo

11th November 2005

Abstract

TLSF is a bounded-time ($O(1)$), Good-fit allocator. TLSF is implemented using a combination of the Segregated lists and bitmaps data structures.

This document presents a detailed description of the TLSF allocator. The data structures and the key internal algorithms are explained.

1 Description

The TLSF data structure can be represented as a two-dimension array. The first dimension splits free blocks in size-ranges a power of two apart from each other, so that first-level index i refers to free blocks of sizes in the range $[2^i,2^{i+1}[$. The second dimension splits each first-level range linearly in a number of ranges of an equal width. The number of such ranges, $2^{\mathcal{L}}$, should not exceed the number of bits of the underlying architecture, so that a one-word bitmap can represent the availability of free blocks in all the ranges. According to experience, the recommended values for $\mathcal{L}$ are 4 or, at most, 5 for a 32-bit processor. Figure 1 outlines the data structure for $\mathcal{L} = 3$.

TLSF uses word-size bitmaps and processor bit instructions to find a suitable list in constant time. For example, using the *ffs*[1] instruction it is possible to find the smaller non-empty list that holds blocks bigger or equal than a given size; and the instruction *fls*[2] can be used to compute the $\lfloor \log_2(x) \rfloor$ function. Note that it is not mandatory to have these advanced bit operations implemented in the processor to achieve constant time, since it is possible to implement them by software using less than 6 non-nested conditional blocks (see glibc or Linux implementation).

Given a block of size $r > 0$, the first and second indexes (fl and sl) of the list that holds blocks of its size range are: $fl = \lfloor \log_2(r) \rfloor$ and $sl = \left\lfloor (r - 2^{fl})/2^{fl-\mathcal{L}} \right\rfloor$. For efficiency reasons, the actual function used to calcualte sl is $\left\lfloor r/s^{fl-\mathcal{L}} \right\rfloor - 2^{\mathcal{L}}$. The function mapping_insert computes efficiently fl and sl:

[1] *ffs*: Find first set. Returns the position of the first (least significat) bit set to 1.

[2] *fls*: Find last set. Returns the position of the most significant bit set to 1.

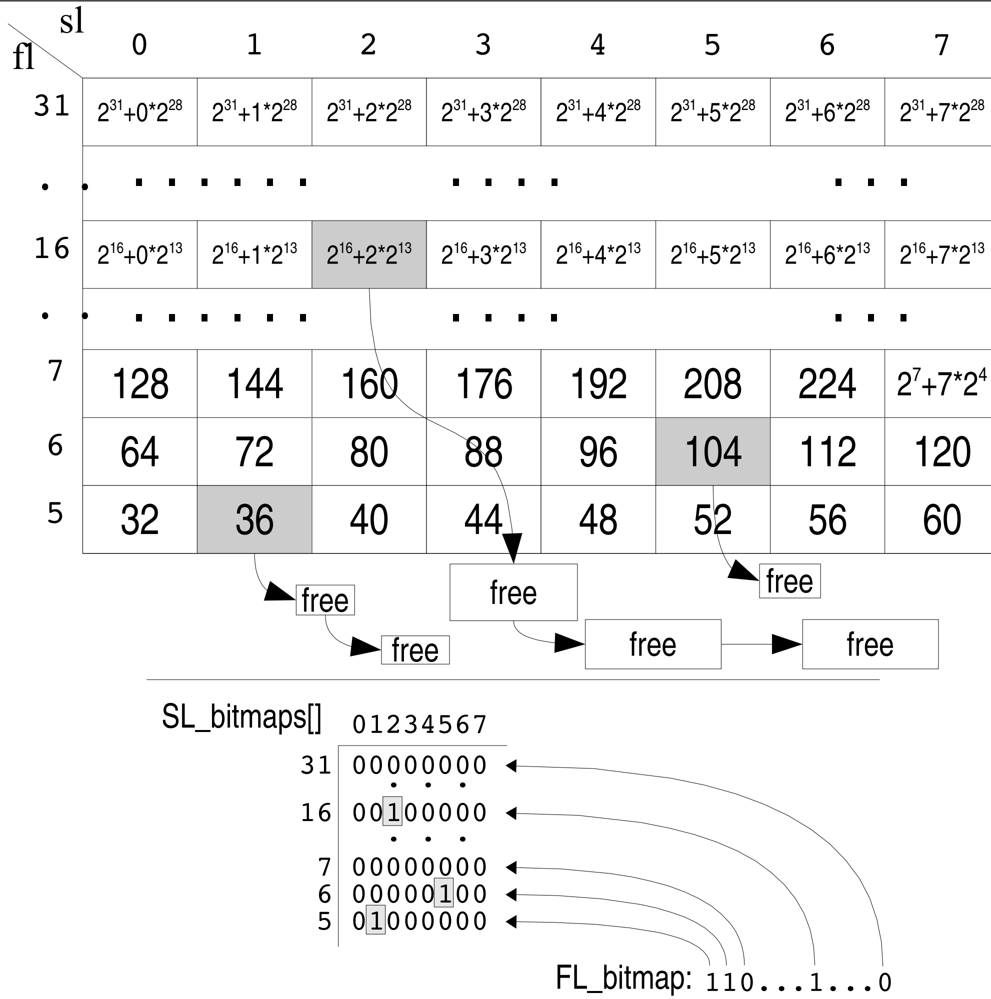

Figure 1: TLSF data structures example.

```
procedure mapping_insert (r: integer; fl, sl: out integer) is
begin
   fl   := fls (r);
   sl   := (r right_shift (fl - ℒ)) - 2^ℒ ;
end mapping_insert;
```

For example, given the size $r = 74$, the first level index is $fl = 6$ and the second level index is $sl = 1$. The binary representation of the size gives an intuitive view of the values of fl and sl:

$$\mathtt{r} = 74_d = \overset{15\,14\,13\,12\,11\,10\,9\,7\,\mathbf{6}}{0\,0\,0\,0\,0\,0\,0\,0\,0\,1}\;\underbrace{\overset{\mathtt{fl=6}}{\overset{5\,4\,3}{0\,0\,1}}}_{\mathtt{sl=1}}\;\overset{2\,1\,0}{0\,1\,0}_b$$

The list indexed by $fl = 6$ and $sl = 1$ is where blocks of sizes in the range [72..80[are

located. But if the requested size is 74 and we search in this list, then we have to discard blocks of sizes 72 and 73, which will introduce an additional and unpredictable time to the algorithm. Instead of discarding smaller blocks, TLSF will start searching from the list of blocks whose minimum size is at least as large as the requested size. In the case of the example, it will start in $fl = 6$ and $sl = 2$, i.e., the list holding blocks of sizes [80..88[. This decision makes TLSF a Good-fit, rather than a Best-fit policy. The function mapping_search computes the values of fl and sl used as starting point to search a free block.

```
procedure mapping_search (r: in out integer; fl, sl: out integer) is
begin
   r    := r + (1  left_shift (fls(r) - L) ) - 1;
   fl   := fls (r);
   sl   := (r right_shift (fl - L)) - 2^L;
end mapping_search;
```

Note that the mapping_search function rounds up the requested size to the closest list. On one hand rounding up blocks causes internal fagmentaion, but on the other hand it greately reduces other types of fragmentation: external and *structural*. The reader is redered to other publications of the autors for a detailed explanation of this issue.

Now, the function search_suitable_block finds non-empty list that holds blocks larger than or equal to the one pointed by the indexes fl and sl. This search function traverses the data structure from right to left in second level indexes and then upwards in first level, until it finds the first non-empty list. Again, the use of bit find instructions allows to implement the search in a very compact manner.

```
function search_suitable_block (fl, sl: in integer) return address is
begin
   temp_bitmap    := SL_bitmaps[fl] and ( FFFFFFFF_16 left_shift sl );
   if temp_bitmap ≠ 0 then
      non_empty_sl :=ffs (temp_bitmap[fl]);
      non_empty_fl :=fl;
   else
      temp_bitmap    := FL_bitmap and (FFFFFFFF_16 left_shift (fl+1));
      non_empty_fl   := ffs (temp_bitmap);
      non_empty_sl   := ffs (SL_bitmaps[non_empty_fl]);
   end if;
   return head_of_list (non_empty_fl, non_empty_sl);
end search_suitable_block;
```

By following the example, the returned free block is the one pointed by the list $(6, 5)$ which holds blocks of sizes [104..112[.

```
function Malloc (r: in integer) return address is
begin
    mapping_search (r, fl, sl);
    free_block := search_suitable_block (r, fl, sl);
    if not(free_block) then return error;
    remove_block (free_block);
    if size (free_block) > split_size_threshold then
        remaining_block := split (free_block);
        mapping_insert (size(remaining_block), fl, sl);
        insert (remaining_block, fl, sl);
    end if;
    return free_block;
end Malloc;
```

```
procedure Free (block: in address) is
begin
  merged_block := merge_left (block);
  merged_block := merge_right (merged_block);
  mapping_insert (size(merged_block), fl, sl);
  insert (merged_block, fl, sl);
end Free;
```

The *Free* function always tries to coalesce neighbour blocks. Merge_left checks whether the previous physical block is free, if so, it is removed from the segregated list and coalesced with the block being freed. Merge_right does the same operation but with the next physical block. Physical neighbours are quickly located using the size of the free block (to locate next block) and a pointer to the previous one, which is stored in the head of the freed block. The cost of all operations is constant ($O(1)$).

2 Download

The code is available at: Real-Time Dynamic Storage Allocation Site