Tunable Pool Allocator

Zipline firmware challenge

Solution by

Seif HADRICH

linkedin

Why do you need this allocator ?

When you want to allocate a dynamic data, you have basically 2 options

Option 1 : Use malloc/free to allocate and deallocate data from the heap.

Pro :
It's very flexible, off-the-shelf solution, works with all sizes: The heap size can be extend when needed (of course if there is a free same space in the RAM).

Cons :
Not very efficient, the run time is not constant => Not suitable for real time application.

Option 2 : Use Pool Allocator : It has a constant run time so more suitable for real time application.

Pro :
When tuned correctly it can allocate and deallocate data in a very efficient way.
Constant and short run time, constant footprint and no segmentation.

Cons :
It's not designed to work for all applications, it's designed to work efficiently with specific needs.

How to use this pool allocator ?

1. Initialization :

```
// Returns true on success, false on failure.

bool pool_init(const size_t* chunk_sizes, size_t chunk_size_count);
```

Inputs :
1. **chunk_sizes** : Array of all different allocation sizes you will need to allocate in the run time.
2. **Block_size_count** : Number of different sizes.

How to pass the initialization phase :

1. Should not insert redundant sizes.
2. Chunck_size_count **<= Param_Max_Nb_Chunk**.
3. All sizes are within : [**Param_Min_Chunk_Size_8b**, **Param_Max_Chunk_Size_8b**].
4. Only used 4 bytes aligned size (the size has to be multiple of 4).
5. Don't use null size.
6. Make sure there is enough memory space.

2. Allocation :

```
// Returns pointer to allocate memory on success, NULL on failure.

 void* pool_malloc(size_t n);
```

Similar to malloc().

3. Deallocation :

```
// Release allocation pointed to by ptr.

void pool_free(void* ptr);
```

Similar to free().

How to tune it?

The pool allocator is designed to be efficient and it could not be efficient if it's not correctly tuned.
Before jumping to the tuning let's see how it works.

How does this alloctor work ?

The allocator uses blocks and chunks. The block contains many chunks.
The chunks is the piece of data where the size is known, (entered by the user at init phase).
The allocator reserved one big block of memory and spit it into many small chunks of the same size.
So one specific block for every different size.
When the user asks for allocation, the program will look at the corresponding block and get one chunk from it.
When a chunk is deallocated the program will save it into a corresponding linked list (one linked list for each different size)
When no more shunk is available (linked list empty and all chunks in the block are used) the pool_allocator will try to get a new block from the heap and again splits it into chunks.

At the initialization phase, the program allocates a new block for every size, splits the block into chunks and initializes the linked lists.

Allocation : pool_alloc()

When pool_alloc() is called, the allocator process this steps :

Step 1 : Check if the corresponding linked list has a chunk.

If yes : Remove this chunk from the linked list and return it to the user.
If no : Move to **step 2**.

Step 2: Check if there is a free chunk in the linked list of the corresponding size.

If yes : Remove it from the linked list and return it to the user.
If no : Move to **step 3**.

Step3 : Try to get a new block from the heap.

If yes : Split this new block into chunks, add them to the linked list and jump back to **Step 2**.
If no : Return NULL.

Deallocation : pool_free()

Step 1 : Use the user input chunk address to find the chunk header. (the chunk header is 4 bytes data stored at address = chunk_address - 4 bytes)

Step 2 : Get chunk id : The header is made up of 4 bytes :
* 1 bytes to store the chunk id.
* 3 bytes to store the relative address.

Step 3 : Add the chunk to the corresponding linked list.
(The chunk is inserted at the beginning of the linked list (root of the linked list : LIFO)

Tuning parameters :

```
27
28  /*
29  ____________________________________________________________________________
30  */
31  //Params :
32  #define Param_Heap_Size_8b          65536   // 64 KB RAM < MAX_HEAP_SIZE_8b6
33  #define Param_Coef                  8       // < MAX_COEF
34  #define Param_Max_Nb_Chunk          6       // < MAX_NB_CHUNK
35  #define Param_Max_Chunk_Size_8b     1024    // < MAX_CHUNK_SIZE_8b
36  #define Param_Min_Chunk_Size_8b     4       // > MIN_CHUNK_SIZE_8b
37
38  /*
39  ____________________________________________________________________________
40  */
```

- **Param_Heap_Size_8b :** The heap size in bytes.
- **Param_Coef** : The block size equal to (**Param_Coef** X **max_chunk_size)** **(max_chunk_size** : the maximum size entered by the user, so it's a user input and not a parameter.)
- **Param_Max_Nb_Chunk :** The maximum allowed number of chunk sizes the user can enter.
- **Param_Max_Chunk_Size_8b :** The maximum allowed size the user can enter.
- **Param_Min_Chunk_Size_8b :** The minimum allowed size the user can enter.

Maximum value : (cannot be changed)

The maximum and minimum allowed values of the corresponding parameters.

```
/*
_____________________________________________________________________________
*/
/*
absolute max values please don't change these values
*/
#define MAX_HEAP_SIZE_8b            0x00FFFFFF
#define MAX_COEF                    0xFFFF
#define MAX_NB_CHUNK                0b00111111
#define MAX_CHUNK_SIZE_8b           0x000FFFFF
#define MIN_CHUNK_SIZE_8b           1

/*
_____________________________________________________________________________
*/
```

Good tuning practices:

Set the **Param_Heap_Size_8b** so you have enough RAM space for the dynamic data but don't waste too much space and prevent others from using it.(RAM = heap + stack + global variables)

Set the **Param_Coef** to choose the block size, try to set it in a way to limit the frequency of allocating new blocks in the run time.

Set **Param_Max_Nb_Chunk** equal or at least not very bigger than the max size chunk you plan to use. (This helps not only saving space but also detecting corrupted data).

Set **Param_Max_Nb_Chunk** equal or at least not very bigger than the number of different chunk sizes you want to use.(This helps not only saving space but also detecting corrupted data).

Developper :

Project architecture and files.

pool_alloc.cpp : allocator source code + tuning parameters
pool_alloc.h : allocator header

Source code :

Address :
In order to improve the code efficiency I used 4 bytes alignment data.
I used relative addresses rather than absolute RAM addresses.
In the program the relative address is called **position.**
xx_size_32b : means how many 4 bytes.
xx_size_8b : means how many bytes.

chunk_size_32b = chunk_size_8b X **4**

The linked list :
Rather than using a separate data structure, the address of next node is actually stored in the value of the address of the current node :
(**node -> next** is **heap[node]**) so we only need to separately store the roots of the linked list, this is stored in the heap.

The **chunk** is made up of header + data.
The allocator returns the address of the data which is the next address of header + 4 bytes

header_adr_32b = chunk_adr_32b - **1**

Debugging :

To help debugging i implemented some debugging functions, only available when the debugging feature is activated. (#define DEBUG_POOL_ALLOC)

RAM dump :

I am also using a specific 4 bytes data to help understanding the memory dump:

0xEEAAEEBB : Means probably never used data (all data initialized like that)
0xDEADBEEF : Means probably shouldn't use this adr (it's padding used for alignment chunk % block)
0xDAxxxyyy : Data set by the uint_test while xxx is the size of chunk and yyy is a counter.

Unit testing :

I did check the flowing edges cases :

- Not enough of available chunks so the allocator gets a new block to create more chunks.
- Not enough of free space to make a new block.
- Free the whole allocated data and use it again.
- Use the pool_alloc() and pool_free() when the pool_alloc() didn't succeed or called
- Set wrong params (don't respect the min and max values)
- Allocated with a random address (out of the heap)
- Free invalid pointer (out of the heap)
- ...

What next ?

If i have time i would probably spend it on the following features :

Testing :

- Check more edge cases.
- Implement a unit testing module to automate testing.
- Compile it with the real compiler (probably C/C++ for **TMS570**)
- Test the code in HIL (I only test it on my 64B computer with **clang++**)

Security :

- Detect if for example the free was called twice without any allocation in between.
- Detect if the data / address is coherent : prevent crashing when data is corrupted.

Effetienty :

- When allocating a new block in the run time, don't split it totally at the first call in chunks : That could take a relatively long time when probably only a few chunks will be useful.

Evaluate performance :

Implement a performance library to check :

- Heap memory useless space.
- Run time : Min Max average.
- Nb new added blocks in the run time.
- Nb of useless chunks.
- Nb of padding (alignment chunks/ block).
- ...

Learning / Source:

I quickly consulted some sources to learn about pool_allocator and malloc but didn't have time to dive on this subject.

Sources i consulted for this challenge :

- https://en.wikipedia.org/wiki/Memory_pool
- https://www.youtube.com/watch?v=74s0m4YoHgM
- http://dmitrysoshnikov.com/compilers/writing-a-pool-allocator/#:~:text=A%20Pool%20allocator%20(or%20simply,blocks%20of%20a%20predefined%20size.