## Design decision and changes to the MMU

The MMU was initially design to be a linked list with the following structure:



Given this data structure, we would have an runtime for request\_memory\_block**()**, and release\_memory\_block**().**

### First Change to the MMU

We soon realize that given this structure, for the 128kb sized blocks we would yield to the maximum of 256 blocks. Which the MMU would first take around 4 blocks. In addition, given a linked list data structure, it would be hard to debug if we ever ran into problems.

Therefore we redesigned the MMU to use an array: char lookup\_table**[**256**]** . This is a 1 to 1 mapping for the memory blocks. For example lookup\_table**[**0**]** would represent the block of 0x10008000 , the char value in the lookup table would be either 0 or 1. 0 would be mean this block is free and 1 would mean the block is being used some process.

The new implementation would be slower than the linked list implementation. Having an array, this makes our request and release memory block to run in time.

In addition to the MMU’s lookup table we also needed other variables, by the end of project 1, the MMU is as follows.



When the OS starts, a function of mmu\_init would be ran to initialize the MMU.

* unsigned int free\_mem would be initialized to Image$$RW\_IRAM1$$ZI$$Limit
* unsigned int max\_memwould be initialized to 0x10008000
* unsigned int actual\_sizewould be calculated based on the actual amount of blocks we would have upon runtime. This would be the actual size of the lookup table. Any data over this limit would be not used.
* char memory\_available is the flag to indicate that if there is any more memory to be requested. This flag is initialized as false

### Second Change to the MMU

The second change in the MMU occurred near the end of the project. That is, when process A, B, and C were implemented, we came to the realization that, since both kernel and user are requesting memory from the same pool of memory, once we are out of memory, kernel functions will not able to function. In addition, in some rare cases, buffer overflow (memory invasion) occurs.

To overcome (reduce), the buffer overflow problem, we increased the memory block size from 128 B to 256 B. By doubling our block size, we observed that we significantly decreased the number of hard fault we would encounter. With this observation, we realized that our previous hard faults were most likely due to buffer overflow.

However, increasing the memory block size did not solve the problem which when we run out of memory, every process that requires memory, including kernel, would get blocked. To overcome this, we separated the user stack and kernel stack. The user memory is then limited to 32 blocks, where the kernel would have whatever is left. By running the OS on the ARM board with the debugger, it seems like kernel would have around 80 blocks to use. This is implemented in such a way that the user would have the top 32 blocks (0x10008000 to 0x10008000 - 31\*256), while the kernel would have the rest, however, we decided to limit it to 64 blocks so that in case of future need of special memory allocation we have some to spare.

By the end of this change, our MMU took the following structure.



Notice we introduced the user\_stack\_table[32], as well as user\_max\_mem, user\_min\_mem. These variables would keep track of the user stack table as well as the top and bottom of the user stack. In addition, the memory\_available flag is also modified so that it only reflects the user\_stack\_table.

With the addition of user stack, and since the user processes would need to call the old request and release memory blocks, we introduced two new function calls for the kernel processes:

* void**\*** k\_request\_kernel\_memory\_block**()**
* void**\*** k\_release\_kernel\_memory\_block**()**

These functions guaranteed that our KCD, wall clock and other kernel functions would still function and not get blocked on memory when there is no more memory for user.