CS-518 - UMALLOC

Ashwin Patil (aap327) | Sammed Admuthe (ssa180)

1 Introduction

To the operating system, memory is nothing but an array of bytes. All the other functionalities such as permissions to access the memory block, memory heap/stack are nothing but policies imposed on that array of bytes. Allocating memory to a process at runtime is referred to as Dynamic Memory Allocation. In Linux, dynamic memory allocation can be achieved by using malloc() or calloc() functions. While both functions allocate memory dynamically, calloc() ensures that memory allocation is contiguous. The job of the memory allocator is to take in the size of memory needed as input and find the block of memory that satisfies the given conditions and return a pointer to the first byte of this block of memory. In addition, the memory allocator will also have to maintain some metadata with each allocation in order to manage (allocate/free) the block of memory. In this project, we have implemented our own “malloc” function called “umalloc” that imitates the functionality of malloc on a predefined fixed-size memory array. We have also implemented our own “free” function called “ufree” that deallocates the memory previously assigned.

2 Procedures

2.1 umalloc

In this section we have explained all the methods/procedures used in implementing our own “malloc” called “umalloc”.

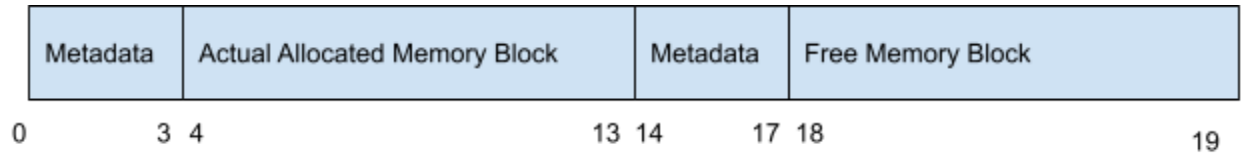
Every time we call umalloc, the very first thing we check is if the memory is initialized. If the memory is not initialized then we initialize the memory we want to manage.
Array of bytes that is 10MB in size.?

Input to umalloc is the “size” of memory that we need to allocate. If all the conditions are met, umalloc sets aside the required amount of memory along with its metadata and returns the address of the first byte of that memory allocation.

For each memory allocation, we store the metadata first, and right after where the metadata ends, we store the actual memory.
Example: Let’s assume we have a total memory of 20 bytes. Let’s assume that our metadata is 4 bytes per allocation. Now if we want to allocate 10 bytes of data, we first store the metadata in

the first 4 bytes and after that, we store the actual memory allocation. So in total, we use 14 bytes of memory.

Fig:



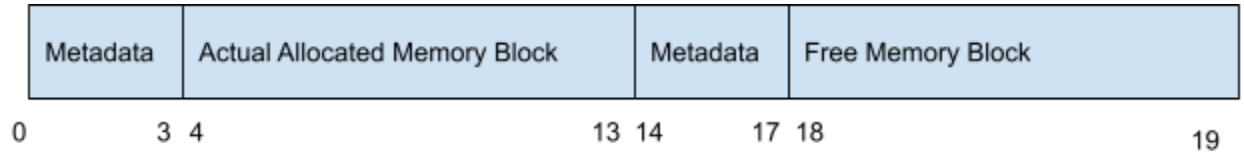
How umalloc finds the right place within the array for memory for allocation is explained further in the memory allocation strategy 3.1
What is metadata? What are its attributes? and how it is used to iterate through the array of bytes (memory) is explained further in 4.1

2.1.1 next

So as we can see how the predefined fix-sized array that we call memory looks like. Metadata followed by actual data. We need to have a method to iterate over the “memory” that we have. We make use of the “next” function for this purpose. First we need to understand what information “Metadata” actually holds. Metadata holds the information about the memory allocation. If the memory allocation is valid or not. If the memory allocation is valid, what is the size of that memory allocation. Two attributes: 1) “used” - is the memory allocation in use or not 2) “blocksize” - the offset that indicates the size of memory allocation.

To iterate over all the allocations in our memory, all we need to do is take the first address in the memory (which will always be a metadata since all the actual memory allocations are after the metadata), look at the block-size in that metadata, and move further in memory by blocksize. This will take us to the next Metadata present in the memory. If we perform the next operation and we get a NULL in return, that means we are iterating outside the memory range that we have defined.

Example:



Here we are assuming that total memory is 20 bytes. There is one memory allocation of 10 bytes already performed. Metadata_0 holds the following information:
Used = ‘t’
Blocksize = 10
Now if we perform next(0) operation, we will jump to the next metadata which is located at location 14.

2.1.2 allocate

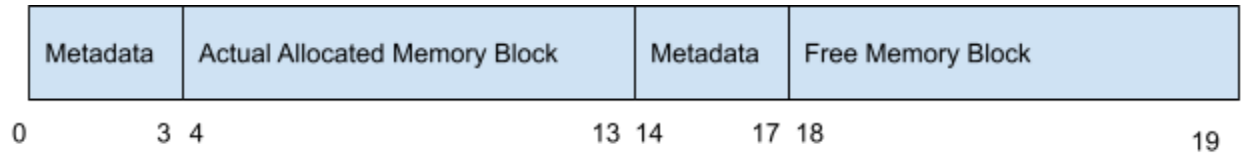
When we call the “umalloc” function with some size as input, umalloc finds the first-fit position in the memory that can be allocated and calls the “allocate” function. Allocate function compares available size and requested size and calculates remainder size. Based on the value of remainder size, allocate decides if there is a need to create a new metadata that can accommodate one more allocation of at least 1 bytes size.

Example:

Initial state

umalloc(10)

State after umalloc(10)



Allocate performs following operations :
Requested size = 10
Available size = 16
Remainder = 16-10 = 6
This remainder size can accommodate a new allocation of 2 bytes max, since in this remainder size it also has to accommodate the size of metadata.

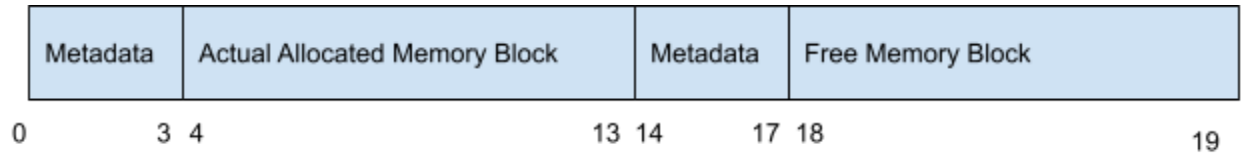
2.2 ufree

In this section we have explained all the methods/procedures used in implementing our own “free” called “ufree”.

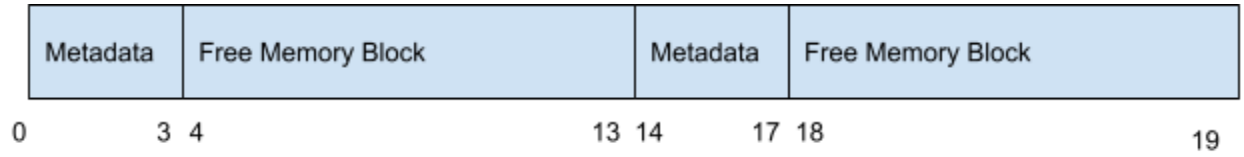
Any memory that is dynamically allocated using “umalloc” can be freed (made available for use again) using “ufree”. This “ufree” function takes the address of actual memory allocation as input and changes the status of ‘used’ attribute in metadata to ‘f’, indicating that particular memory allocation is no longer in use.

But we have to check for more conditions that we need to handle. If there are contiguous free blocks forming after a free operation then we need to combine all these free blocks into one single free block.

Example:



Now if we perform free operation on the first memory allocation, we get two contiguous free memory blocks.



Since we are now wasting space by having two Metadatas while we only need one to represent this space. For this purpose we call the combine function explained below.

2.2.1 combine

In the “combine” function, we start analyzing the memory blocks from the beginning. If we come across any metadata that indicates a free memory block, we check its next metadata, if the next metadata also indicates a free memory block, we increase the size of the first free block by the size of the encountered free block along with the size of the metadata (combine the two contiguous free memory blocks). If the next metadata indicates a memory block that is in use, then we go on looking for the next metadata that is free and perform the same comparisons again. We do this till we reach the end of memory.
Following flowchart should explain the logic behind the combine function.

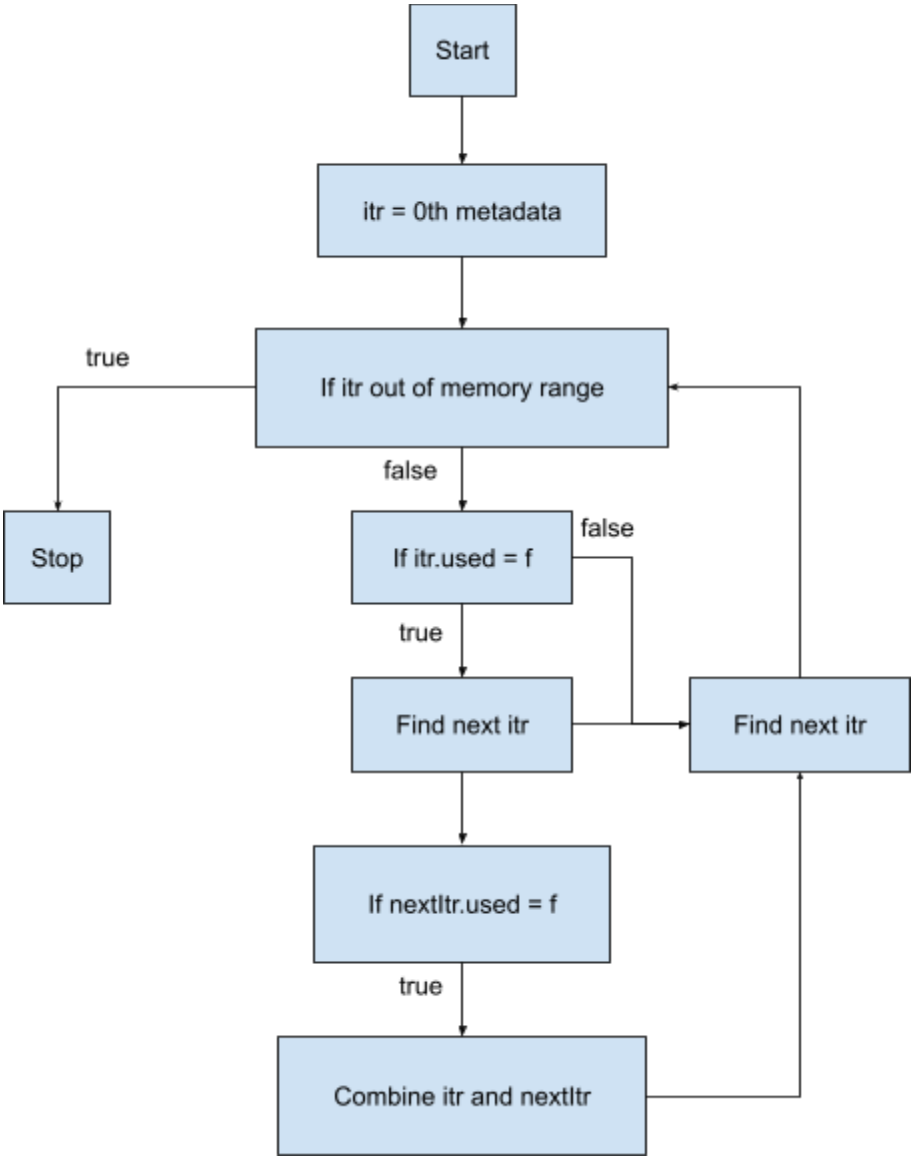


Fig : Flowchart representing logic behind combine function

3 Memory Allocation Strategy

In this section, we have explained the memory allocation strategy used by umalloc function. So far as we have seen, memory allocations are nothing but metadata followed by actual memory block allocation. Whenever we call “umalloc” function, umalloc needs to find a memory block that is free and that can satisfy the required size. The strategy for finding this memory block large enough to satisfy the incoming request is explained below.

3.1 First Fit

Logic:-

We start iterating memory from the 0th metadata. If the metadata indicates that the memory block is in use, we hop onto the next metadata. Else if the metadata indicates that the memory block is not in use, we check if the ‘blocksize’ indicated by that metadata is large enough to accommodate the incoming request. If it cannot satisfy the requirement, we hop onto the next metadata that indicates the next memory block not in use. Else if the requirements are satisfied, we return the address of that memory block. We do this until we reach the end of the memory.

Implementation:-

Following flowchart diagram should explain the logic behind the first-fit memory allocation strategy.

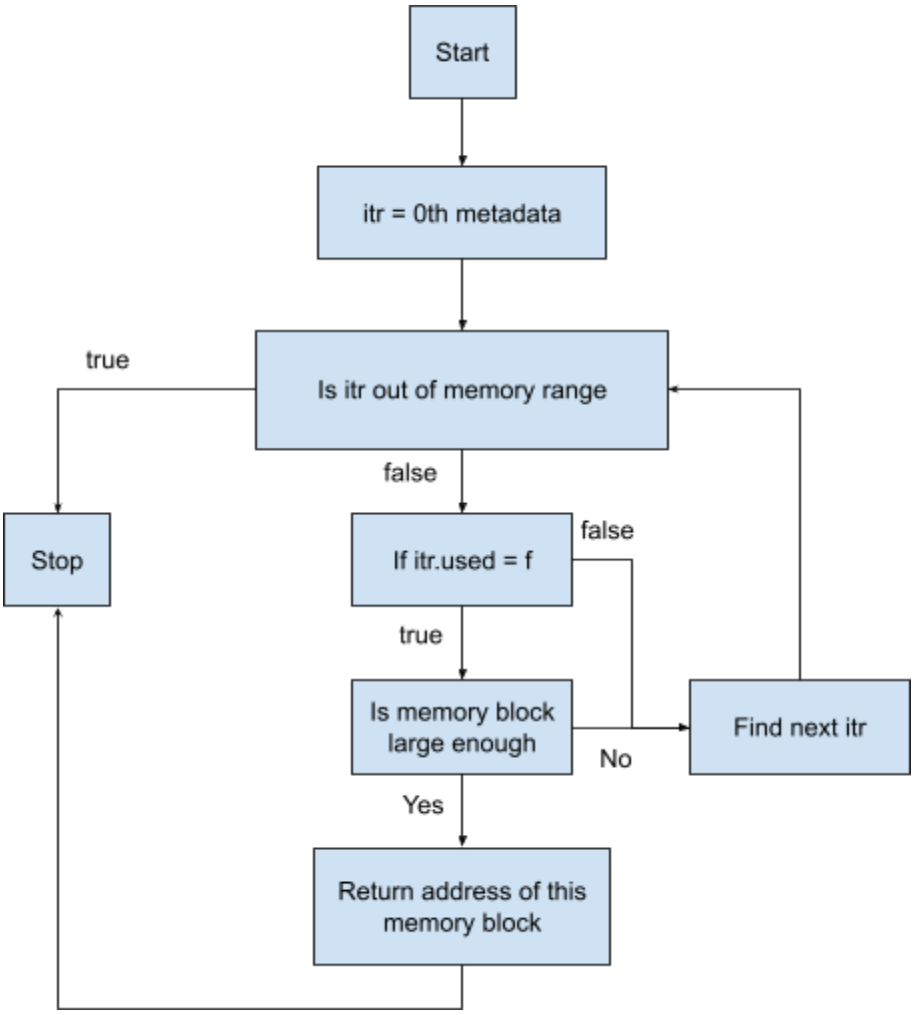


Fig : Flowchart representing logic behind First-Fit memory allocation strategy

4 Structs

In this section we have explained structs that we have used to store all the information about metadata.

4.1 Metadata

```
typedef struct Metadata
{
        unsigned int blocksize; //size of the memory allocation
        char used;              //if the memory allocation is in use or not
} Metadata;
```

5 Test Case Evaluation

In this section, we have evaluated the given test cases.
Note : In screenshots you can see that we have made calls to “malloc” function. We have a MACRO written umalloc.h file which replaces the “malloc” calls with “umalloc” and “free” calls with “ufree”.

5.1 Consistency

Initially, the whole memory is empty. And after the memory initialization there is only one metadata present indicating the remaining memory is not in use.
So let’s say we perform a memory allocation of size “x”, we get back the “address” of the allocation in return. Now if we write to this memory and free it. We will go back to the state after memory initialization. Now if we try to allocate memory of size “x” again, we should get back the same address in return. In the end we free this memory allocation.

5.2 Maximization

In this test, we allocate 1 byte of memory. If successful, we double the size (2 bytes) and try again. We keep on doing this till we get a NULL. Let’s say we get a NULL on umalloc(X). We try to allocate the half size of X. If we get a success, we free the allocated memory and return the value of X.

Note : At one point in execution of this test case, we try to allocate memory that is larger than the total memory available. We expect the console to display one error message at this point.

5.3 Basic Coalescence

In this test, we perform 2 allocations. First allocation should be half of the size of maximal allocation that we computed in 5.2. Second allocation should be ¼ th the size of maximal allocation. If both the allocations are successful, the test case is passed otherwise the test fails.

5.4 Saturation

In this test, we fill up our entire memory of 10MB with small allocations.
First we fill up 9 MB of memory with 9216 1KB (1024B) allocations.
We saturate the rest of the memory with 1B allocations.

5.5 Time Overhead

We perform this test right after the test mentioned in 5.4. In this test, we free the last allocation we performed. Now the memory is completely saturated except at the very end of the memory. Now if we try to allocate the same amount of memory at that location, it will be successful but it will take some time since we have to traverse the whole memory till the end. We take note of the time required to perform this last allocation. The most recent run gave the time overhead of 643843 nano-seconds.

5.6 Intermediate Coalescence

We perform this test right after we perform the test mentioned in 5.5. In this test, we free all the allocations that we have performed earlier. This will result in an empty array of memory. Now we try to allocate the memory calculated in test 5.2.

```
aap327@cp:~/Desktop/A1$ make
gcc -g -c memgrind.c -L../ -lumalloc
gcc -g -c umalloc.c
ar -rc libumalloc.a umalloc.o
ranlib libumalloc.a
gcc memgrind.o umalloc.o -o memgrind
aap327@cp:~/Desktop/A1$ ./memgrind

********* Consistency Test Start *********

Address of first allocation 0x55bcdd2d3068
Address of second allocation 0x55bcdd2d3068
Consistency test PASS!

********* Consistency Test End *********

********* Maximal Allocation Test Start *********

memgrind.c : 16 Error: Memory overflow!!
Maximal allocation is 8388608

********* Maximal Allocation Test End *********

********* Basic Coalescence Test Start *********

Allocated half memory
Allocated additional quarter memory
Freed allocated memory
Max allocation done
Basic Coalescence test success

********* Basic Coalescence Test End *********

*********Saturation Test Start *********

Fragment size = 0
memgrind.c : 107 Error: No free Memory available
Total number of allocations = 117532

********* Saturation Test End *********

********* Timeoverhead Test Start *********


Time for 1B allocation in saturated mem : 643843 nano-seconds

********* Timeoverhead Test End *********

********* Intermediate Test Start *********

Freed allocated Memory
Maximum Memory allocated
Intermediate Coalescence Test Success!

********* Intermediate Test End *********

Overall Test Time: 75494 milliseconds
aap327@cp:~/Desktop/A1$
```

6 Conclusion

Using a custom memory allocator, we were successfully able to test for consistency, maximization, basic and intermediate coalescence. We get a time overhead of 643843 nano-seconds which is comparable to the time overhead for the original malloc library. In addition, the custom error handling enables users to determine the exact cause of malloc and free misuse.

While implementing umalloc and ufree, unutilized fragments could be generated because in certain cases umalloc returns more memory than requested by the user. This becomes prominent when we try to allocate less memory than actual block size in freed memory space and adding additional memory and metadata in remaining free space is not feasible which in turn creates unused fragments.