Robert Alvarez, Emiliano De La Cruz & Diana Garcia
HW 2 – October 31st, 2022

Overall project

SWE decisions:

We make the following structs to organize our information:

Block:

size_t bytes_used: Number of consecutive free bytes starting at the location of where bytes_used is allocated to the last free mapped byte.

Block *next_block: Pointer to the next block inside of the map base on its memory address.

Map:

Block *first_block: Pointer to the first block inside the map base on its memory address. If all the bytes inside the map had been used, it points to NULL.

Block *largest_block: Pointer to the block inside map with the largest amount of consecutive free bytes. This pointer is always accurate and if there are no block it points to NULL.

Map *next_map: Points to the next map base on memory address. Last map points to NULL.

List:

Map *first_map: Points to the first map base on the memory address of its location. If no map has been map, it points to NULL.

Map *largest_map: Points to the map with the block that have the largest amount of free consecutive bytes. This pointer doesn't always have to be accurate, there may be a map with a larger block, but the pointer would always be to an existing map. If no map has been map, it points to NULL.

A global List variable was created, LL, to organize all the mappings made my mmap(), Map. It initially set the pointers to NULL and after a Map variable is created it is added into LL. We have a function call make_map which mmap a page and use the first sizeof(Map) of the mapping to make a Map variable to be added into the LL and the rest of page is use to make a single block which start where the map start + sizeof(Map).

Problems encounter:

Making a pointer to a List was undesired because we need to make a mmap with the sizeof(List) if it didn't exist, so to reduce the number of mmaps we make a List variable as List LL = {NULL, NULL}.

__malloc_impl

SWE decisions:

We check that the requested size is at least a size of a pointer so we can use the pointer of Block (Block *next_block) so we can optimize the number of bytes used. If the size is less than the pointer we request sizeof(Block) where the first sizeof(size_t) is used to store the number of bytes used by the ptr, and the use the other sizeof(Block)-sizeof(size_t) to return to the user. If the user request more than a size of a pointer, we need requested_size + sizeof(size_t). If the need_size + sizeof(Map) is greater than a page we need to mmap it individually because for every regular mmap for a page we use sizeof(Map) before getting the first block so that way we know that it wouldn't fit if we send need_size to be allocated from a regular malloc where a Map is needed. If we can get a block for the need_size, make a map and still have some bytes left, we ask for a ptr by calling get_allocation(need_size). get_allocation() is explained later on this report.

Problems encounter:

Some edge cases are when the user request a size of 0, we return NULL. The major edge case was the one mention above where the requested size and a map is more than a page which would make an infinite loop of mmaps.

__calloc_impl

SWE decisions:

First check that the result of multiplying calloc parameters can be store in a size_t variable. If it can't save it in a size_t or if the result is 0, we return NULL. Otherwise, we call malloc with the result and save the returned pointer. Base on the number of bytes allocated for the pointer, we iterate through it and set every byte to '\0' and we return the pointer.

Problems encounter:

By setting the bytes to '\0' without type casting it to an unsigned char it has some side effects of not setting all the bytes properly.

__realloc_impl

SWE decisions:

We check the same thing as on malloc about having a required size of at least sizeof(Block *), if it isn't we need only a size of block. If the required size is more than that we set the needed size to be the required size + sizeof(size_t). If the size needed + sizeof(Map) is more or equal to a page, we make an individual mapping. If the individual mapping was not made, we continue with the size needed. If the pointer passed was mapped individually, we get a new pointer with the need size, copy the minimum number of bytes in between the size of the pointer passed and the needed size, we free the pointer passed (either by munmap if individual mapping was made or by calling add_block()), and returned the new pointer.

Problems encounter:

Some edge cases: If the user give a NULL pointer as a parameter, we call malloc with the required size and return that pointer. If require size is 0, we free the pointer given and return NULL.

__free_impl

SWE decisions:

Given the pointer, we see based on the number of bytes used if it was an individual mmap we munmap it. Otherwise, we send the pointer to add_block().

Problems encounter:

No problems encounter on the free function but add_block had some challenging parts. You can read about them letter on this report.

Extra functions

make_map

SWE decision:

We call mmap with the size of a page, use the first sizeof(Map) for the map and the rest for a block. Once the map is made and the block is set, we add the map into LL base on its address memory place and finally return the map variable created.

Problems encounter:

This is a third function which is almost completely isolated, so we didn't have problems base on this function.

add_block_to_map

SWE decision:

A block and a map, where the block belong to, are passed as parameters. We iterate over the blocks on the map until we found the last block that is before the one passed as parameter. Once the previous block was found we update the pointers, so the block passed is inside the list of blocks. Then we check if we can merge the blocks, we can check that by seeing if the location of the block + the bytes_used is the same as the address of the block in front. If it is the same, we update the current pointer to what the next block is pointing to and add the sizes into the current block. Once the merges are done, we return 0 if successful. For an unsuccessful result, see the problems encounter section.

Problems encounter:

If a pointer with overlapping memory (usually happen if the same pointer is sent to be free more than once) is found, a -1 would be return because we can't have overlapping blocks.

update_largest_map

SWE decisions:

We iterate over all the maps inside LL to update LL.largest_map with the largest map base on the number of bytes of its largest block on the map.

Problems encounter:

A problem rarely encounter was a segmentation fault due to an assumption that a map or a block within a map exist.

add_block

SWE decisions:

Given a block, we find the map where it belongs to and call the function add_block_to_map() with the map and block. If the function was unsuccessful, we return, otherwise we check if everything from the map has been free. If it is the case, we modify LL to remove the map from the list before munmaping the map.

Problems encounter:

Some issues came up depending on if the map was the largest, first, the only one or a combination of the previous options. Each one is easily catch with a few if statements. Another problem was the reallocation of the map into LL if the munmap was unsuccessful.

update_largest_block

SWE decisions:

Given a map, we iterate over all the block inside of it to set map->largest_block base on the amount of consecutive free bytes on a block.

Problems encounter:

Some segmentations faults were gotten at the beginning because of a wrong assumption that a block must always exist withing a map.

get_allocation

SWE decisions:

Given the number of bytes to get, we look into the largest_map inside LL. If largest_map don't have enough we call update_largest_map() and check again. If no map have a block with enough bytes, we call make_map() and use the return map to make two block, one with the bytes requested to be return and the other one with the remaining amount of bytes. If a map do have a block with enough bytes we check if we can enough bytes to return a block and make another block from the remaining. If it is possible to make another block, we get the number of bytes requested from the end of the block, so we don't have to update pointers, just the bytes_used of each. If no other block can be made, we search for the block pointing to the largest block, update pointer to remove largest_block from map, update new largest_block and return the block that just got taken out.

Problems encounter:

Assuming that a map must contain a block without checking that it wasn't NULL. Another issue was that we didn't always check that a block can be made after getting the number of bytes requested which make side effects on overwriting spaces in memory. Lastly, that we didn't always update the largest_block for every case that the function could take.

Testing

A lot of testing was made on the main.c file. You can run it on the terminal by writing "make clean main" and run it can be run as "./main". Specific edge cases for free NULL pointers, allocating more than a page, sum of allocation is more than a page, checking that calloc set every character to '\0' and that the realloc function free the space of the previous pointer and return a new one with the proper amount of characters copied was given in testing(). But the final test was made by a random function that make n function calls combining 4n/5 malloc calls, n/5 calloc calls, n/7 realloc calls and n free calls. The value of n is the parameter of the function and we recommend to use it with no more than 25000 because of the time it takes but you are free to change it as wanted. The random allocation code is as follows:

```
void **ptr = (void **) malloc(n*sizeof(void *));
size_t page = (size_t) getpagesize();
size_t size[n];
size_t count[(n/5)+1];

size_t n_bytes;
char c;
char *p;

printf("\tALLOCATING:\n");
for (size_t i = ((size_t) 0); i < n; i++) {
  //print call number if it is divisible by 1000
  if (i % (n/20) == 0) {
    printf("i = %zd\n", i);
  }
  //Set the amount of memory to request
  size[i] = ((size_t) rand()) % page;
  //If the function call is divisible by 5 we use calloc
  if (i % 5 == 0) {
    count[i/5] = ((size_t) rand() % 10);
    size[i] /= 5;
    ptr[i] = __calloc_impl(count[i/5], size[i]);
    n_bytes = count[i/5]*size[i];
  }
  //Otherwise we use malloc
  else {
    ptr[i] = __malloc_impl(size[i]);
    n_bytes = size[i];
  }
  //If the function call is divisible by 7, we call realloc
  if (i % 7 == 0) {
    //If we are on the second half, do realloc at the pointer minus half of the list maximum number
    if (i > (n/2)) {
      size[i-(n/2)] = ((size_t) rand()) % page;
      ptr[i-(n/2)] = __realloc_impl(ptr[i-(n/2)], size[i]);
    }
    //If we are on the first half, do realloc at the pointer
    else {
      size[i] = ((size_t) rand()) % page;
      ptr[i] = __realloc_impl(ptr[i], size[i]);
```

```
      n_bytes = size[i];
    }
  }

  //Place random characters inside the recently allocated pointer
  c = ((char) ((rand() % 96) + 32));
  p = ((char *) ptr[i]);
  while (n_bytes--) {
    *p++ = c;
     if (c >= ((char) 127)) {
       c = ((char) 32);
     } else {
       c++;
     }
   }
}

//printf("After allocating memory:\n");
//print_LL_info();

printf("\tFREEING:\n");
for (size_t i = ((size_t) 0); i < n; i++) {
  if (i % (n/20) == 0) {
    printf("i = %zd\n", i);
  }
  __free_impl(ptr[i]);
}

printf("After freeing all the memory:\n");
print_LL_info();

free(ptr);
```

The output when n = 10,000 is as follows:

ALLOCATING:
i = 0
i = 500
i = 1000
i = 1500
i = 2000
i = 2500
i = 3000
i = 3500
i = 4000

i = 4500
i = 5000
i = 5500
i = 6000
i = 6500
i = 7000
i = 7500
i = 8000
i = 8500
i = 9000
i = 9500
FREEING:
i = 0
i = 500
i = 1000
i = 1500
i = 2000
i = 2500
i = 3000
i = 3500
i = 4000
i = 4500
i = 5000
i = 5500
i = 6000
i = 6500
i = 7000
i = 7500
i = 8000
i = 8500
i = 9000
i = 9500
After freeing all the memory:

Inside print_LL_info()
LL.largest_map = 0x0
(End of function) Exiting print_LL_info()

This random_allocation() function use two helper functions, one is check_calloc(void *ptr, size_t size) which iterate ptr over size bytes and check that all its content is set to '\0'. If any byte is not set to '\0', it print the index and the number inside that index. Another helper function used is to print the maps and blocks inside LL which is called print_LL_info() inside alloc.c. The function looks as follows:
676 void print_LL_info()

```
{
  printf("\nInside print_LL_info()\n");
  printf("LL.largest_map = %p\n", LL.largest_map);

  Map *m = LL.first_map;
  Block *b;
  size_t tot;
  while (m != NULL) {
    printf("map = %p\n", m);
    b = m->first_block;
    tot = ((size_t) 0);
    while (b != NULL) {
      printf("block = %p\n", b);
      printf("block->bytes_used = 0x%zx, block->next_block = %p\n", b->bytes_used, b->next_block);
      tot += b->bytes_used;
      if (b == b->next_block) {
        printf("\nERROR: b == b->next_block, b = %p\n\n", b);
        break;
      }
      b = b->next_block;
    }
    printf("map->largest_block = %p\n", m->largest_block);
    printf("Total bytes available = 0x%zx\n", tot);
    if (m == m->next_map) {
      printf("\nERROR: m == m->next_map, m = %p\n\n", m);
      break;
    }
    m = m->next_map;
  }
  printf("(End of function) Exiting print_LL_info()\n\n");
}
```

More testing function simulating bash commands malloc, calloc, realloc, and free calls where implemented inside main.c.