System Programming - Linker Lab Report

2019 Spring, 2017-18570 Sungchan Yi

* I followed the **C99 standard** for the implementations for this lab.

0 Introduction

At first glance, there was too much to do. I had to read all the files and resolve all the symbols by myself. Then I started to realize that the lab pdf forewarned me that I would have to read other people's codes.

So I looked at `#define`s, `#include`s, the test cases, and all kinds of `.h` files. I also noticed that I couldn't change the code for the test cases. Moreover the compile options did not specify anything for the linker.

Thus I decided to try run-time interpositioning of dynamic libraries.

1 Part One

After seeing the test cases, I had to override `malloc`, `free`, `calloc`, and `realloc` function. So my first try was writing the code for `mymalloc` and `myfree`. I tested it inside the `init` constructor, and it worked well.

Now the hard part was how to make the functions of test cases call the functions I defined.

Since my functions would be a strong symbol, I thought the linker would choose my function instead of the original implementation in `libc`. But the test case does not `#include <malloc.h>` that I wrote, so it seems that the test cases are just calling the original implementation.[1]

To try and find why my functions weren't being called, I added `printf` statements for debugging purposes. Then the program started not to execute properly and kept giving me segmentation faults. So I pondered for a few hours on how to fix this.[2] I even tried including my own header file `<malloc.h>` into the test case, but it still didn't work.

After many tries, I found out that the `printf` statement added for debugging caused segmentation faults. When I erased it, it just worked like magic. So I made minor changes and after rearranging functions a bit, I implemented all 4 functions.

The procedure was quite simple. Call `dlsym(RTLD_NEXT, func_name)` and save it to a function pointer. Call the function from the pointer and return. During the process, track down the changes in allocated bytes, freed bytes.

---

[1] I could not change the compiler options, and test case source codes. I did use the `dlsym` function but it didn't work for some unknown reason.

[2] At this point of time, I didn't know `printf` was an unsafe function.

2 Part Two

Before I began part 2, I read the implementation of `memlist`. A list with dummy head `list` is created in `init()`, so every time memory is allocated, I realized that I had to add a new item to the list.

I also read the methods defined in `memlist.h`, and the implementations in `memlist.c`. I found out a few things on how this `memlist` works. Every time an memory allocating function is called, I call `alloc()` function a new item will be added to the list. If the item is a duplicate, the reference count will automatically increase. Also, every time memory is freed, I call `dealloc()` and the reference count will decrease.

I also got the idea of using the reference count to track down non-deallocated memory. At `fini()`, I would traverse the list and print any item with reference count greater than or equal to 1.[3]

There weren't that many changes when I was extending my implementation from Part 1. I just had to keep track of `n_freeb` to print at the end.

3 Part Three

At first, I had no idea on how to print function's name and the offset. So I searched on Google and tried to gather information about memory allocation, instruction pointers and etc. During the process, I found out that `realloc(ptr, size)` frees allocated memory in `ptr` and *reallocates* memory of size `size` somewhere else.[4] So I had to modify Part 2 a bit.

I was stuck here for a few days, and kind of forgot about this. When I tried to do the lab, I had to look up the lab pdf again. There I found out some useful functions that would help me finish this lab. They were the functions in the `libunwind` library.

I went to google and searched for `libunwind` and read the documentation for the functions that were mentioned in the lab pdf. Additionally I looked at some example codes and ran it on the test cases and found out how I should implement the lab.

`unw_step` function helped me step out of the stack frame and `unw_get_proc_name` was a function to get the name of the procedure. The main thing to consider was how many times I should call the `unw_step` function. The example code for `libunwind` let me know that I had to call `unw_step` 3 times.[5] Furthermore, the offset was biased to be 5 more, so I had to subtract 5 for the offset. This was because the `call` instruction was 5 bytes.

Here is how I implemented the `get_callinfo` function.

```
#include <string.h>

```

[3]Reference count `cnt` is initialized to 1 so if it is freed after allocation, `cnt` would have to be less than 1.

[4]Well, to be exact, `realloc` *can* reallocate memory on the same place as `ptr`.

[5]This was another problem, since if the `main` routine called a function `foo` and if `malloc` was called inside of `foo`, I would have to call `unw_step` once more. Later, I was notified that there were no such test cases.

```c
int get_callinfo(char *fname, size_t fnlen, unsigned long long *ofs)
{
    char f_name[fnlen]; // stores function name
    unw_cursor_t cursor; // cursor to the stack frame
    unw_context_t context;
    unw_word_t off; // offset
    unw_proc_info_t pip; // proc info pointer
    char procname[256]; // procedure name

    // Store the name of the caller function
    char mainf[] = "main";
    char libcmain[] = "__libc_start_main";
    int ret;

    if(unw_getcontext(&context)) return -1; // Get context error
    if(unw_init_local(&cursor, &context)) return -1;
    for(int i = 0; i < 3; i++) { // Perform step 3 times
        if(unw_step(&cursor) < 0) return -1;
        if(unw_get_proc_info(&cursor, &pip)) break; // get proc info
        ret = unw_get_proc_name(&cursor, procname, 256, &off);
        // get name and offset
    }
    // call instruction takes up 5 bytes
    *ofs = (unsigned long long) off - 5;
    ret = unw_get_proc_name(&cursor, procname, 256, &off);
    if(strcmp(procname, libcmain) == 0) {
        // if the procedure name is equal to __libc_start_main
        // copy main to fname
        strcpy(fname, mainf);
    } else { // else copy the current procedure name to fname
        strcpy(fname, procname);
    }
    return ret;
}
```

4 Bonus Part

The bonus part was quite easy, since the reference count would be less than 0 if a double free occurs. But I had segmentation faults for a while, because I put the `if(block == NULL)` statement after checking double free. That statement was for checking an illegal free, which means that the `find` function in `memlist` would return `NULL`. Referencing `cnt` of `NULL` would of course give segmentation faults. Switching the order of statements fixed the problem.

5 Wrap-Up

I did this lab quite early after it was announced. So after a few days, there were so many questions on eTL and I had a hard time reading all the questions and answers. Eventually, I had to modify some of my implementations. And suddenly an email came and it said that I could choose to implement in C99 standard. Since I originally implemented everything in C99 standard, I thought there was nothing to modify. For the last time, I reviewed my code and tried some of my own test cases to check if it worked properly.

Then I saw an interesting question on eTL, and it said that there will be some special cases. Due to the sudden changes, I restarted the whole thing. So I looked up for more references and found out the behaviors of memory allocating functions for special cases.

Here's some things I found out.

- `free(NULL)` does nothing.[6]
- Calling `realloc` on freed memory pointer works. (Undefined Behavior)
- `realloc(NULL, size)` is the same as `malloc(size)`.

And moreover, I read the behavior of `realloc` on a reference[7], and it said these two things about how reallocation is done.

(a) Expanding or contracting the existing area pointed to by `ptr`, if possible. The contents of the area remain unchanged up to the lesser of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.

(b) Allocating a new memory block of size `new_size` bytes, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block.

I only implemented (b), so I fixed my code to work for (a). The comments show how I did it.

```c
void *realloc(void *ptr, size_t size) {
    char* error;
    reallocp = dlsym(RTLD_NEXT, "realloc");
    if((error = dlerror()) != NULL) {
        fputs(error, stderr);
    exit(1);
    }
    void* res = reallocp(ptr, size); // first, reallocate
    LOG_REALLOC(ptr, size, res); // LOG it
    n_realloc++; // increment count

    if(res == ptr) { // pointer didn't change resizing happened
```

[6] It crashes on some OS, but it worked on `sp1.snucse.org`

[7] `https://en.cppreference.com/w/c/memory/realloc`

```
        item* block = find(list, ptr); // find block
        // this block cannot be null
        if(block -> size > size) { // decreased size
            n_freeb += block -> size - size;
        } else { // increased size
            n_allocb += size - block -> size;
        }
        // update totals
        dealloc(list, block); // deallocate
        block -> cnt --;
        alloc(list, res, size); // allocate item in list
    } else { // freed memory in ptr and allocated somewhere else
    item* block = find(list, ptr); // try to find ptr in list
    /*
        There is a chance block might be NULL
        Case 1. ptr is NULL
            In this case, realloc(NULL, size) was called.
            C standard says that this is equal to malloc(size)
        Case 2. ptr is not NULL but not found in the list
            In this case, realloc to some other location was called.
            The processor will still try to allocate memory.
            This is undefined behavior.
        In both cases, free doesn't happen
    */
    if(ptr == NULL || block == NULL) {
        alloc(list, res, size); // allocate item in list
        n_allocb += size; // add
    } else { // block was found
        n_freeb += block -> size; // original place was freed
        n_allocb += size; // add
        dealloc(list, block); // deallocate
        block -> cnt --;
        alloc(list, res, size); // add item in list
        }
    }
    return res;
}
```

6 Conclusion

Although so many questions and answers confused me on how I should implement the `realloc` function, the lab itself was quite interesting. I really like the idea of interpositioning, and I think it really would be useful for debugging purposes. Interpositioning at run-time really blew my mind.