

M1522.000800 System Programming, Fall 2017  
Linker Lab: Memtrace  
Assigned: Mon, Oct 16, Due: Mon, October 30, 13:59

## Introduction

In this lab, we use the at-load-time library interpositioning to trace calls to the dynamic memory management system of a compiled binary program. Your memory tracer will be able to log all dynamic memory management calls of a program, and identify memory blocks that a program does not free. We then use a low-level libraries to identify the call sites that allocate the unfreed memory.

If you are still not satisfied, you can augment your memory tracer to prevent incorrect memory deallocations of random binaries in an optional bonus question.

By solving this lab, you will learn a lot about library interpositioning and simple management of data within a shared library. A lot of the code is provided already, this lab will also force you (and teach you how) to read someone else's code. We will also make use of a debugger library that will show you how to pinpoint exact program points in your running programs.

The lab may require a significant effort if you are not familiar with C programming. We provide enough hints and support in the lab sessions so that everyone can solve the lab with a good mark. Nevertheless, start early and ask early!

# Logistics

## Forking a repository (5 points)

In *git* terminology, a *fork* is an exact copy of a repository. This allows users freely perform changes to a source base without modifying the original project. This method is widely used in the open source community and is referred to as the *fork and pull* model: multiple collaborators share a base repository yet, not all of them have the right to modify it. Developers must first *fork* the base repository, make some changes and perform a *pull request*. The base repository administrator can review each pull request and decide if it can be *merged* into the base repository or not.

In this lab, we will practice part of this concept. We will distribute the handout for this lab as a read-only repository on our class *gitlab* server ([git.csap.snu.ac.kr](http://git.csap.snu.ac.kr)) at the following address:

<http://git.csap.snu.ac.kr/sysprog17/linklab>

After you have logged in, you can proceed to *fork* the *linklab* repository from the given link. You can *fork* a repository in two ways, via the web interface or via the command line.

### Forking a repository from the web-interface

1. direct your browser to the repository you want to fork.
2. click the “*fork*” button (in our case it is located under the repository title, towards the left side)

Now, you should have a exact copy of the link lab handout on your personal git account. As we learned before you can now proceed to *clone* this repository on your local machine:

```
devel@gentoo ~ $ git clone <URL>
```

## Compiling, Running and Testing

Makefiles in the various directories assist you with compiling, running, and testing your implementations. Do not modify the Makefiles unless explicitly instructed. Run `make help` to find out which commands the Makefile supports.

A set of test programs is provided with which you can test your solution. Note that we will use a different test set to evaluate your submission..

```
devel@gentoo ~/linklab/part1 $ make help
```

## Submission

To submit your solution simply *commit*, and *push* the changes to your remote repository.

```
devel@gentoo ~/linklab $ git add <list of modified files>
devel@gentoo ~/linklab $ git commit -m <commit message>
devel@gentoo ~/linklab $ git push origin master
```

We won't consider any *commit* which timestamp is greater than the lab's deadline. If for some reason you cannot *push* to the git server and there are 5 minutes before the deadline then, and only then, create a *tar* file of your submission and send it via email to the TAs.

# Dynamic Memory Management

## Overview

In many languages, memory is explicitly allocated and sometimes even deallocated by the programmer. The POSIX standard defines the following four dynamic memory management functions that we want to intercept.

**void \*malloc(size\_t size)**

malloc allocates `size` bytes of memory on the process' heap and returns a pointer to it that can subsequently be used by the process to hold up to `size` bytes. The contents of the memory are undefined.

**void \*calloc(size\_t nmemb, size\_t size)**

calloc allocates `nmemb*size` bytes of memory on the process' heap and returns a pointer to it that can subsequently be used by the process to hold up to `size` bytes. The contents of the memory are set to zero.

**void \*realloc(void \*ptr, size\_t size)**

realloc changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents are copied up to `min(size, old size)`, the rest is undefined.

**void free(void \*ptr)**

free explicitly frees a previously allocated block of memory.

Use the Unix manual to learn more about malloc, calloc, realloc, and free:

```
devel@gentoo ~/linklab $ man malloc
```

Dynamic memory management functions are defined in the standard library. Include `stdlib.h` in your program to have access to `malloc`, `calloc`, `realloc`, and `free`.

## Example

The following example program demonstrates how dynamic memory management functions can be used.

```
#include <stdlib.h>

void main(void) {
    void *p;
    char *str;
    int *A;

    // allocated 1024 bytes of memory
    p = malloc(1024);

    // allocated an integer array with 500 integer
    A = (int*)calloc(500, sizeof(int));

    // allocate a string with 16 characters...
    str = (char*)malloc(16*sizeof(char));

    // ...then resize that string to hold 512 characters
    str = (char*)realloc(str, 512*sizeof(char));

    // finally, free all allocated memory
    free(p);
    free(A);
    free(str);
}
```

example1.c

Note that all allocators return an untyped pointer (`void*`) that needs to be converted to the correct type in order to prevent compiler warnings.

## Part 1: Tracing Dynamic Memory Allocation (20 Points)

Subdirectory: part1/

In this part, your job is to trace all dynamic memory allocations/deallocations of a program. Print out the name, the arguments, and the return value of each call to `malloc`, `calloc`, `realloc`, and `free`. When the program ends, print statistics about memory allocation (number of bytes allocated over the course of the entire program, average size of allocation).

For the program `test1.c` given below

```
#include <stdlib.h>

void main(void) {
    void *a;

    a = malloc(1024);
    a = malloc(32);
    free(malloc(1));
    free(a);
}
```

test1.c

the following output should be generated (the pointer values may differ from system to system):

```
devel@gentoo ~/linklab/part1 $ make run test1
[0001] Memory tracer started.
[0002]      (nil) : malloc( 1024 ) = 0xb87010
[0003]      (nil) : malloc(  32 ) = 0xb87420
[0004]      (nil) : malloc(   1 ) = 0xb87450
[0005]      (nil) : free( 0xb87450 )
[0006]      (nil) : free( 0xb87420 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total        0
[0012]
[0013] Memory tracer stopped.
devel@gentoo ~/linklab/part1 $
```

Leave the statistics on the total number of freed memory bytes (`freed_total`) at 0 for now.

You can start from scratch or implement your solution by extending the skeleton provided in `~/linklab/part1/memtrace.c`. You do not need to modify `callinfo.c/h` for this part.

Use the logging facilities provided in `~/linklab/utils/memlog.c/h`. This will ensure that your output looks exactly as above and we can automatically test your submission for correctness.

## Part 2: Tracing Unfreed Memory (20 Points)

Subdirectory: part2/

The tracer from part 1 is quite useful, but it has a serious shortcoming: it cannot check whether all allocated memory has been freed. In this part, we add this functionality. While the program is running, keep track of all allocated blocks and check which ones get deallocated. When the program ends, print a list of those blocks that were not deallocated. In addition, also compute and output statistics about the total number of freed memory bytes.

For the program `test1.c` shown on the previous page, the output should look as follows:

```
devel@gentoo ~/linklab/part2 $ make run test1
[0001] Memory tracer started.
[0002]          (nil) : malloc( 1024 ) = 0x2415060
[0003]          (nil) : malloc( 32 ) = 0x24154c0
[0004]          (nil) : malloc( 1 ) = 0x2415540
[0005]          (nil) : free( 0x2415540 )
[0006]          (nil) : free( 0x24154c0 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total         33
[0012]
[0013] Non-deallocated memory blocks
[0014]   block                size      ref cnt   caller
[0015]   0x2415060            1024        1      ??? : 0
[0016]
[0017] Memory tracer stopped.
devel@gentoo ~/linklab/part2 $
```

You need to keep track of blocks to implement this functionality. We provide a memory block list in `~/linklab/utils/memlist.c/h` so that you don't have to write a linked list yourself (of course, you can write your own if you like). The documentation can be found in the header file.

Extend your tracer from part 1 by copying the source file `memtrace.c` into the directory of this part. You do not need to modify `callinfo.c/h` for this part.

**Note:** In the case of `realloc`, in order to correctly compute the total number of allocated/freed bytes you can consider `realloc` as consisting of two steps: a deallocation step immediately followed by an allocation step.

## Part 3: Pinpointing Call Locations (20 Points)

Subdirectory: part3/

To be truly useful, we need to know where in the program code the memory allocations/deallocations occur. Implement this functionality by printing the function name and the offset of the instruction calling the dynamic memory management function.

The logging functions already print the PC using the function `get_callinfo()` defined in `callinfo.h` and implemented in `callinfo.c/h`.

```
// return the PC of the callsite to the dynamic memory management function
//
//  fname      pointer to character array to hold function name
//  fnlen      length of character array
//  ofs        pointer to offset to hold PC offset into function
//
// returns
//   0         on success
//  <0        on error
//
int get_callinfo(char *fname, size_t fnlen, unsigned long long *ofs);
~/.linklab/part3/callinfo.h

int get_callinfo(char *fname, size_t fnlen, unsigned long long *ofs)
{
    return -1;
}
~/.linklab/part3/callinfo.c
```

Currently, the implementation is empty and returns -1 indicating error, which is why the PC is shown as "(nil)" in the previous outputs. Your job is to fill it with useful content so that it returns the name and offset of the caller to the `malloc/calloc/realloc/free` library function.

The output of `test1.c` should now look as follows:

```
devel@gentoo ~/.linklab/part3 $ make run test1
[0001] Memory tracer started.
[0002]      main:6 : malloc( 1024 ) = 0x14f0060
[0003]      main:10 : malloc( 32 ) = 0x14f04c0
[0004]      main:1d : malloc( 1 ) = 0x14f0540
[0005]      main:25 : free( 0x14f0540 )
[0006]      main:2d : free( 0x14f04c0 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total         33
[0012]
[0013] Non-deallocated memory blocks
[0014]   block      size      ref cnt   caller
[0015]   0x14f0060   1024        1      main:6
[0016]
[0017] Memory tracer stopped.
devel@gentoo ~/.linklab/part3 $
```

Indeed, we see that our memory tracer now pinpoints the exact locations of the calls by looking at the disassembly of `~/linklab/test/test1`

```
devel@gentoo ~/linklab/test $ objdump -d test1
...
00000000004004a0 <main>:
 4004a0: 53                      push    %rbx
 4004a1: bf 00 04 00 00         mov     $0x400,%edi
4004a6: e8 e5 ff ff ff       callq   400490 <malloc@plt>
 4004ab: bf 20 00 00 00         mov     $0x20,%edi
4004b0: e8 db ff ff ff       callq   400490 <malloc@plt>
 4004b5: bf 01 00 00 00         mov     $0x1,%edi
 4004ba: 48 89 c3               mov     %rax,%rbx
4004bd: e8 ce ff ff ff       callq   400490 <malloc@plt>
 4004c2: 48 89 c7               mov     %rax,%rdi
4004c5: e8 96 ff ff ff       callq   400460 <free@plt>
 4004ca: 48 89 df               mov     %rbx,%rdi
4004cd: e8 8e ff ff ff       callq   400460 <free@plt>
 4004d2: 31 c0                 xor     %eax,%eax
 4004d4: 5b                      pop     %rbx
 4004d5: c3                      retq
 4004d6: 66 2e 0f 1f 84 00 00   nopw    %cs:0x0(%rax,%rax,1)
 4004dd: 00 00 00
devel@gentoo ~/linklab/test $
```



## Bonus: Detect and Ignore Illegal Deallocations (+10 Points)

Subdirectory: bonus/

*Note:* to work on the bonus assignment, you have to complete part 2 first.

Some programs call free more than once on a memory block.

```
#include <stdlib.h>

void main(void) {
    void *a;

    a = malloc(1024);
    free(a);
    free(a);
    free((void*)0x1706e90);
}
```

test4.c

This programming error results in the process being aborted as shown below:

```
devel@gentoo ~/linklab/test $ ./test4
*** Error in `./test4': double free or corruption (top): 0x000000025da010 ***
===== Backtrace: =====
/lib64/libc.so.6(+0x72603) [0x7f09c84f4603]
/lib64/libc.so.6(+0x77ee6) [0x7f09c84f9ee6]
...
Aborted
devel@gentoo ~/linklab/test $
```

Extend your tracer from part 2 by detecting and ignoring deallocations of not allocated memory blocks. For the program test4.c shown above, the output should look as follows:

```
devel@gentoo ~/linklab/bonus $ make run test4
[0001] Memory tracer started.
[0002]      main:6   : malloc( 1024 ) = 0x1b30060
[0003]      main:11  : free( 0x1b30060 )
[0004]      main:19  : free( 0x1b30060 )
[0005]      *** DOUBLE_FREE *** (ignoring)
[0006]      main:23  : free( 0x1706e90 )
[0007]      *** ILLEGAL_FREE *** (ignoring)
[0008]
[0009] Statistics
[0010]   allocated_total      1024
[0011]   allocated_avg       1024
[0012]   freed_total        1024
[0013]
[0014] Memory tracer stopped.
devel@gentoo ~/linklab/bonus $
```

The output above was generated with a completed part 3. You can work on the bonus question without solving part 3 first; in that case the output will show (nil) for the program locations.

## Report (+15 points)

In the *linklab* repository you will find a file called `report_template.{docx/odt}` use this file as a template to write your report for this lab. The report is divided into five main sections, you should fill each of these sections and submit the report in printed form in the submission box before the deadline.

1. **Introduction:** short description of the lab objectives, and what your report will discuss.
  2. **Part 1 to Part 3:** description of the steps taken in order to solve the relevant part. You should include a subsection of any problems you encounter during this part, and what you did to solve it. Remember that simply copy-and-pasting screenshots of code is not sufficient.
  5. **Experience with *gitlab*:** Short paragraph mentioning your thoughts on using *gitlab* as a means to distribute and submit homework and laboratories.
- ➔ **Bonus:** (optional, include it only if you implement the bonus part) description of the steps taken in order to solve this part. You should include a subsection of any problems you encounter during this part, and what you did to solve it.