2019-1 시스템프로그래밍 Lab1

LinkLab

I. Goal of this Lab

Library interpositioning 은 이미 define 되어 있는 library function 에 대해 programmer 가 새롭게 함수를 정의하여 해당 함수를 대체 할 수 있도록 한다. 새로 정의하는 함수에서는 기존 library function 을 호출하는 것 이외에 logging 과 같은 다른 작업을 수행하도록 하여 debugging 및 해당 함수의 tracking 을 용이하게 한다. 이러한 library interpositioning 은 compile time, link time, load/run time 에 일어날 수 있는데 이번 실습에서는 load/run time 에 library interpositioning 을 수행하도록 한다.

이번 Lab 에서 interpositioning 의 대상이 되는 library function 인 함수는 stdlib.h 에 정의 동적할당 함수 즉, malloc, calloc, realloc, free 이다. 동적할당의 경우 (realloc 의 경우) valid 한 pointer 를 주어야 하고 모두 사용한 후에는 free 를 꼭 해주어 memory leak 을 막아야 한다. free 를 해주된는 것은 동적할당 함수를 사용함에 있어 매우 중요하며 할당 받은 해당 memory 가 제대로 free 가 되었는지 확인하는 작업은 중요하다. 이번 실습에서는 동적할당 받은 memory 가 모두 free 가 되었는지, free 가 되지 않은 block 이 있다면 해당 block 은 source code 어느 부분에서 할당 받은 memory 인지 block 의 size 와 함께 확인할 수 있도록 한다.

이 Linklab 은 Part1~3, bonus part 로 구성되어있으며 각 part 를 따라 순차적으로 구현하도록 한다. 우선 Part1 에서는 malloc, calloc, realloc, free 가 test 에서 불릴 때마다 해당하는 메모리 동적할당을 해주고 어떤 argument 를 받아 어떤 값을 return 했는지 log 를 찍어 확인할 수 있도록 한다. test 가 모두 끝난 후에는 전체 allocated byte 수, 평균 allocated byte 수를 계산하여 log 를 찍어주도록 한다. Part2 에서는 Part1 의 내용을 확장하여 free 된 byte 수 또한 마지막에 log 를 찍어주며 free 가 되지 않은 block 이 있는 경우 해당 block 의 address, size, reference count 정보를 찍어주도록 한다. Part3 에서는 메모리 동적할당 함수가 불린 경우 어느 함수의 어느 instruction 에서 불렸는지 함수 이름과 함수 시작점부터 해당 instruction 까지의 offset 을 같이 logging 하도록 한다. 마지막 non-deallocated block 을 찍어줄 때에도 이 정보를 추가적으로 logging 해준다. Bonus part 에서는 illegal 한 free 에 대한 handling 을 추가적으로 구현한다. 이미 free 된 block 의 pointer 를 받는 경우 double-free 에 해당하는 error log 를 출력하도록 하며 invalid 한 pointer 를 받는 경우 illegal-free 에 대한 error log 를 출력하도록 한다.

II. Implementation

i. Part1

Part1 에서는 기본적으로 run-time library interpositioning 을 위한 setting 을 해주어야 한다. 이번 lab 에서 사용하는 original malloc, calloc, realloc, free 함수를 wrapper 함수에서 호출하기 위해서 이들의 original function pointer 를 dlsym 을 통해 가져와 global variable 로 저장해두고 wrapper 함수가 호출될 때마다 이 function pointer 를 이용할 수 있도록 한다. init 함수 내에 이들의 값을 해당 함수 포인터 값으로 초기화하면 이후 wrapper function 에서 original 동적메모리할당 함수를 부를 수 있다.

```
__attribute__((constructor))
void init(void)
{
  //...this code only shows the added part
  // assign stdlib's memory management functions to function ptrs
  if (!mallocp) {
        mallocp = dlsym(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
                fputs(error, stderr);
                exit(1);
        }
  }

  if (!callocp) {
        callocp = dlsym(RTLD_NEXT, "calloc");
        if ((error = dlerror()) != NULL) {
                fputs(error, stderr);
                exit(1);
        }
  }

  if (!reallocp) {
        reallocp = dlsym(RTLD_NEXT, "realloc");
        if ((error = dlerror()) != NULL) {
                fputs(error, stderr);
                exit(1);
        }
  }

  if (!freep) {
        freep = dlsym(RTLD_NEXT, "free");
        if ((error = dlerror()) != NULL) {
                fputs(error, stderr);
                exit(1);
        }
  }
}
```

실질적인 wrapper 함수 구현에서는 global variable 로 저장해 둔 function pointer 를 이용하여 original 함수를 호출하고 log 를 찍을 수 있다. log 의 경우 utils/memlog.h 에 이미 지정되어 있는 macro 를 이용하면 예시 출력과 같은 형태로 출력할 수 있다. 새로 block 을 할당 받는 malloc, calloc, realloc 의 경우 그 size 만큼 n_allocb 변수의 값을 증가시켜주고 n_malloc 과 같은 해당 함수 호출의 cnt 변수를 증가시킨다. free 의 경우 이번 part1 에서는 count 를 하지 않으므로 original free 를 호출하고 log 찍는 것만으로 충분하다.

```
// Replace original stdlib functions with our own functions at run time
void *malloc(size_t size)
{
```

```
  void *ptr;

  ptr = mallocp(size);
  n_allocb += size;
  n_malloc++;
  LOG_MALLOC(size, ptr);
  return ptr;
}

void *calloc(size_t nmeb, size_t size)
{
  void *ptr;

  ptr = callocp(nmeb, size);
  n_allocb += nmeb * size;
  n_calloc++;
  LOG_CALLOC(nmeb, size, ptr);
  return ptr;
}

void *realloc(void* ptr, size_t size)
{
  void *updated_ptr;

  updated_ptr = reallocp(ptr, size);
  n_allocb += size;
  n_realloc++;
  LOG_REALLOC(ptr, size, updated_ptr);
  return updated_ptr;
}

void free(void *ptr)
{
  freep(ptr);
  LOG_FREE(ptr);
  return;
}
```

마지막으로 `statistic` 정보를 출력하기 위해서 `fini` 함수에서 다음의 코드를 수정하였다.

```
LOG_STATISTICS(n_allocb, n_allocb / (n_malloc + n_calloc + n_realloc), 0L);
```

ii. Part2

Part2 에서는 Part1 에서와 달리 free 되는 block 을 count 해야 한다. 최종적으로 free 가 되지 않은 block 이 있는지 여부를 확인하기 위해서는 메모리를 alloc, dealloc 하는 것을 tracking 해야 한다. malloc, calloc 의 경우 utils/memlist.c 에 있는 alloc 함수를 이용하여 새로 allocated 된 block 의 정보를 저장하도록 한다. alloc 을 하는 경우 block 에는 reference count 에 해당하는 정보인 cnt 값이 1 로 setting 된다. 반대로 free 의 경우에는 dealloc 을 불러서 cnt 값을 0 으로 만들어준다. free wrapper function 의 경우 free 함수의 인자로 넘겨받은 포인터 값이 valid 한지 여부를 확인한 후 수행하도록 작성하였다. 즉, 할당되지 않은 block 을 free 하는지 혹은 이미 free 된 block 을 free 하는지 여부를 check 하도록 하였다. realloc 의 경우 dealloc 을 한 후 새로 alloc 을 하도록 하였다. 이때 free 에서와 마찬가지로 dealloc 을 하기 전 pointer 가 valid 한지 확인하도록 하였다. free 에서와 달리 넘겨받은 pointer 가 NULL 인 경우를 예외적으로 handling 하였는데 이는 original realloc 함수에서 pointer 가 NULL 인 경우 error 를 return 하는 것이 아니라 새로운 block 을 할당받는다는 점을 고려하여 handling 한 것이다. 또, realloc 에서 n_freeb 를 계산하는 방식은 “새로 할당 받은 size 가 기존 block 의 size 보다 작은 경우 에만 그 차이만큼 n_freeb 값을 증가시킨다”는 추가 spec 을 반영하여 구현하였다. **(4 번의 공지에 따르는 방법으로 구현)**

```
// Replace original stdlib functions with our own functions at run time
void *malloc(size_t size)
{
  void *ptr;

  ptr = mallocp(size);
  // alloc item to store info of this malloc
  alloc(list, ptr, size);
  n_allocb += size;
  n_malloc++;
  LOG_MALLOC(size, ptr);
  return ptr;
}

void *calloc(size_t nmeb, size_t size)
{
  void *ptr;

  ptr = callocp(nmeb, size);
  // alloc item to store info of this calloc
  alloc(list, ptr, nmeb * size);
  n_allocb += nmeb * size;
  n_calloc++;
  LOG_CALLOC(nmeb, size, ptr);
  return ptr;
}

void *realloc(void* ptr, size_t size)
{
  void *updated_ptr;
  size_t old_size = 0;

  item *item_ptr = find(list, ptr); // find return NULL if there is no item

  if ((item_ptr != NULL) && ((item_ptr->cnt) > 0)) { // valid case
        updated_ptr = reallocp(ptr, size);
        old_size = item_ptr->size;
        if (old_size >= size)   n_freeb += old_size - size;
        LOG_REALLOC(ptr, size, updated_ptr);
        // free the original block first (dealloc doesn't work if block doesn't
exist)
        dealloc(list, ptr);
```

```
  }
  else { // cases that don't need dealloc
        updated_ptr = reallocp(NULL, size);
        LOG_REALLOC(ptr, size, updated_ptr);
        if (ptr == NULL) ;
        else if (item_ptr == NULL) // invalid ptr given
          LOG_ILL_FREE();
        else // when item_ptr->cnt == 0 : already freed
          LOG_DOUBLE_FREE();
  }

  // malloc again for new block
  alloc(list, updated_ptr, size);

  n_allocb += size;
  n_realloc++;
  return updated_ptr;
}

void free(void *ptr)
{
  item *item_ptr = find(list, ptr);
  LOG_FREE(ptr);
  // free when the ptr address is valid
  if ((item_ptr != NULL) && ((item_ptr->cnt) > 0)) {
        n_freeb += item_ptr->size;
        dealloc(list, ptr);
        freep(ptr);
  }
  return;
}
```

statistic 정보를 출력한 이후 free 가 되지 않은 block 인지 여부는 block 의 list 를 따라 돌면서 각 block 의 cnt 값이 양수인지를 확인하면 된다.

```
  LOG_STATISTICS(n_allocb, n_allocb / (n_malloc + n_calloc + n_realloc),
n_freeb);

  item *cur;
  int logged = 0;
  cur = list;
  while (cur != NULL) {
        if (cur->cnt > 0) {
          if (!logged) {
                // Non-deallocated memory blocks part is printed
                LOG_NONFREED_START();
                logged = 1;
          }
          LOG_BLOCK(cur->ptr, cur->size, cur->cnt, cur->fname, cur->ofs);
    }
    cur = cur->next;
  }

  LOG_STOP();
```

iii. Part3

Part3 에서는 malloc, calloc, realloc, free 함수가 어느 함수에서 호출되는지 그리고 해당 함수의 시작점부터의 offset 을 구하는 작업이 필요하다. 이를 구하기 위해서 <libunwind.h>에 있는 unw_get_proc_name 함수를 사용할 수 있다. 이 함수를 이용하면 현재 procedure 의 이름과 offset 을 모두 구할 수 있다. 이 함수를 이용하기 위해서는 우선 malloc 을 부르는 곳, 즉 main 까지 올라가야 한다. 아래 표에 표현한 것 처럼 main 에서 malloc 을 호출하고 malloc 이 alloc 을 alloc 이 get_call_info 를 호출하기 때문에 get_call_info 의 procedure 에서 3 번 올라가야 malloc 을 부르는 main procedure 로 올라갈 수 있다. unw_getcontext 와 unw_init_local 의 경우 unw_get_proc_name 과 unw_step 을 하기 전에 기본 설정을 위해 필요한 수행이다. offp 의 값에서 5 를 뺀 값을 *ofs 의 값으로 넣는 이유는 x86_64 의 call instruction 이 5 byte 로 구성되기 때문이다.

main (test/test*.c)
malloc/calloc/realloc (part3/memtrace.c)
alloc (in utils/memlist.c)
get_call_info (part3/callinfo.c)

```
int get_callinfo(char *fname, size_t fnlen, unsigned long long *ofs)
{
  unw_context_t uc;
  unw_cursor_t cursor;
  long unsigned int offp;
  int i, err;

  err = unw_getcontext(&uc);
  if (err < 0) return -1;
  err = unw_init_local(&cursor, &uc);
  if (err < 0) return -1;

  for (int i=0; i<3; i++) {
        err = unw_step(&cursor);
    if (err < 0)
          return -1;
  }

  err = unw_get_proc_name(&cursor, fname, fnlen, &offp);
  if (err < 0) return -1;

  // difference between real offset and the offp is 5
  *ofs = (unsigned long long) offp - 5;

  return 0;
}
```

iv. Bonus

bonus 에서는 추가적으로 double-free 와 illegal free 에 대해 handling 해준다. 만약 find 를 이용해 찾은 결과값(item_ptr)이 NULL 이라면 넘겨받은 ptr 이 allocated 되지 않은 block 의 주소, 즉 invalid 한 ptr 이므로 LOG_ILL_FREE()를 해준다. item_ptr→cnt == 0 인 경우는 이미 free 된 block 이라는 의미이므로 LOG_DOUBLE_FREE()를 해준다.

```
void free(void *ptr)
{
  item *item_ptr = find(list, ptr);
  LOG_FREE(ptr);
  // free when the ptr address is valid
  if ((item_ptr != NULL) && ((item_ptr->cnt) > 0)) {
        n_freeb += item_ptr->size;
        dealloc(list, ptr);
        freep(ptr);
  }
  else if (item_ptr == NULL)
        LOG_ILL_FREE();
  else // already freed
        LOG_DOUBLE_FREE();

  return;
}
```

III. Result and Additional Test

i. Part1

Part1 에서 구현한 memtrace 를 이용하여 test1 을 실행한 결과는 다음과 같다. allocated_total = 1024 + 32 + 1 = 1057, allocated_avg = 1057 / 3 = 352 로 값이 잘 나옴을 확인할 수 있다. 또한 동적으로 할당받은 주소값을 정확히 넘겨주어 size 가 1 인 block, size 가 32 인 block 을 순차적으로 free 해주는 것을 확인할 수 있다. 이 결과에서 확인할 수 있는 또 다른 사실은 0x213d470 – 0x213d060 = 0x410, 0x213d4a0 – 0x213d470 = 0x30 이라는 것이다. 할당받은 주소값의 차이는 0x410 = 0x400 (1024byte) + 0x10, 0x10 = 0x20 (32byte) + 0x10 을 만족함을 알 수 있다. 0x10 만큼의 차이는 malloc 한 block 에 대한 metadata 를 저장하기 위한 추가적인 공간으로 보여진다.

```
stu114@sp1:~/SP2019/linklab/handout/part1$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]           (nil) : malloc( 1024 ) = 0x213d060
[0003]           (nil) : malloc( 32 ) = 0x213d470
[0004]           (nil) : malloc( 1 ) = 0x213d4a0
[0005]           (nil) : free( 0x213d4a0 )
[0006]           (nil) : free( 0x213d470 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total          0
[0012]
[0013] Memory tracer stopped.
```

ii. Part2

Part2 에서는 non-freed block 을 tracing 하기 위한 구현을 추가적으로 하였다. 이를 위해 utils/memlist.c 에 있는 alloc 함수를 사용하였는데 이 alloc 함수는 동적 메모리 할당이 있을 때마다 calloc 하여 block 을 또 할당받는다. 이로 인해 part1 에서와 주소값 간의 차이가 다르게 나타난다. 0x23344c0 - 0x2334060 = 0x460, 0x2334540 – 0x23344c0 = 0x80 이므로 part1 에서 보다 0x50 만큼 더 차이나는 것을 확인할 수 있다. 이는 item 만큼의 메모리를 더 할당 받기 때문이다. 정확히 sizeof(item)의 값만큼 차이나지 않는 것은 마찬가지로 metadata 의 저장과 alignment 로 인한 것으로 보인다. 또한 Part1 에서는 확인할 수 없었던 freed_total = 32 + 1 과 Non-deallocated memory block 의 정보를 확인할 수 있다.

```
stu114@sp1:~/SP2019/linklab/handout/part2$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]           (nil) : malloc( 1024 ) = 0x2334060
[0003]           (nil) : malloc( 32 ) = 0x23344c0
[0004]           (nil) : malloc( 1 ) = 0x2334540
[0005]           (nil) : free( 0x2334540 )
[0006]           (nil) : free( 0x23344c0 )
[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]   0x2334060          1024       1         ???:0
[0016]
[0017] Memory tracer stopped.
```

아래의 그림에서 calloc 도 정상적으로 동작함을 확인할 수 있다.

```
stu114@sp1:~/SP2019/linklab/handout/part2$ make run test3
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]           (nil) : calloc( 1 , 30785 ) = 0x1dec060
[0003]           (nil) : malloc( 19979 ) = 0x1df3900
[0004]           (nil) : calloc( 1 , 52869 ) = 0x1df8770
[0005]           (nil) : calloc( 1 , 10363 ) = 0x1e05650
[0006]           (nil) : calloc( 1 , 25875 ) = 0x1e07f30
[0007]           (nil) : calloc( 1 , 34045 ) = 0x1e0e4a0
[0008]           (nil) : calloc( 1 , 22514 ) = 0x1e16a00
[0009]           (nil) : calloc( 1 , 59753 ) = 0x1e1c250
[0010]           (nil) : calloc( 1 , 56150 ) = 0x1e2ac20
[0011]           (nil) : malloc( 39816 ) = 0x1e387d0
[0012]           (nil) : free( 0x1e387d0 )
[0013]           (nil) : free( 0x1e2ac20 )
[0014]           (nil) : free( 0x1e1c250 )
[0015]           (nil) : free( 0x1e16a00 )
[0016]           (nil) : free( 0x1e0e4a0 )
[0017]           (nil) : free( 0x1e07f30 )
[0018]           (nil) : free( 0x1e05650 )
[0019]           (nil) : free( 0x1df8770 )
[0020]           (nil) : free( 0x1df3900 )
[0021]           (nil) : free( 0x1dec060 )
[0022]
[0023] Statistics
[0024]   allocated_total      352149
[0025]   allocated_avg        35214
[0026]   freed_total          352149
[0027]
[0028] Memory tracer stopped.
```

realloc 또한 정상적으로 동작하며 test5 의 경우 새로 realloc 하는 block 의 byte 수가 기존 byte 수보다 크기 때문에 realloc 시에는 freed_block 의 값은 변하지 않는다. 마지막 free 시에 free 된 byte 수만 더해져 freed_total 의 값은 100000 이 된다.

```
stu114@sp1:~/SP2019/linklab/handout/part2$ make run test5
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]           (nil) : malloc( 10 ) = 0x1cc4060
[0003]           (nil) : realloc( 0x1cc4060 , 100 ) = 0x1cc40d0
[0004]           (nil) : realloc( 0x1cc40d0 , 1000 ) = 0x1cc4190
[0005]           (nil) : realloc( 0x1cc4190 , 10000 ) = 0x1cc45d0
[0006]           (nil) : realloc( 0x1cc45d0 , 100000 ) = 0x1cc45d0
[0007]           (nil) : free( 0x1cc45d0 )
[0008]
[0009] Statistics
[0010]   allocated_total      111110
[0011]   allocated_avg        22222
[0012]   freed_total          100000
[0013]
[0014] Memory tracer stopped.
```

iii. Part3

Part3 에서는 get_call_info 함수를 구현함으로써 어느 함수의 어느 offset 에서 동적메모리 할당 함수를 호출했는지 tracing 할 수 있다. 이전 part 에서 (nil)로 표현되었던 부분들이 (함수이름):(offset)형태로 표현됨을 확인할 수 있다.

```
stu114@sp1:~/SP2019/linklab/handout/part3$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]         main:6  : malloc( 1024 ) = 0x80f060
[0003]         main:10 : malloc( 32 ) = 0x80f4c0
[0004]         main:1d : malloc( 1 ) = 0x80f540
[0005]         main:25 : free( 0x80f540 )
[0006]         main:2d : free( 0x80f4c0 )
[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]   0x80f060           1024       1         main:6
[0016]
[0017] Memory tracer stopped.
```

실제 instruction 과 비교해보면 정확한 위치를 tracking 함을 알 수 있다.

```
Disassembly of section .text:

0000000000400470 <main>:
  400470:       53                      push   %rbx
  400471:       bf 00 04 00 00          mov    $0x400,%edi
  400476:       e8 d5 ff ff ff          callq  400450 <malloc@plt>
  40047b:       bf 20 00 00 00          mov    $0x20,%edi
  400480:       e8 cb ff ff ff          callq  400450 <malloc@plt>
  400485:       bf 01 00 00 00          mov    $0x1,%edi
  40048a:       48 89 c3                mov    %rax,%rbx
  40048d:       e8 be ff ff ff          callq  400450 <malloc@plt>
  400492:       48 89 c7                mov    %rax,%rdi
  400495:       e8 96 ff ff ff          callq  400430 <free@plt>
  40049a:       48 89 df                mov    %rbx,%rdi
  40049d:       e8 8e ff ff ff          callq  400430 <free@plt>
  4004a2:       31 c0                   xor    %eax,%eax
  4004a4:       5b                      pop    %rbx
  4004a5:       c3                      retq
  4004a6:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4004ad:       00 00 00
```

iv. Bonus

bonus part 에서는 double, illegal free 에 대한 logging 을 해주도록 하였다. 아래 그림에서 처럼 이미 free 한 block 을 다시 free 하려는 경우 Double free 에 대한 error message 를 출력해주며 임의의 invalid 한 주소값을 주는 경우 Illegal free 에 대한 error message 를 출력한다.

```
stu114@sp1:~/SP2019/linklab/handout/bonus$ make run test4
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]         main:6  : malloc( 1024 ) = 0x9d5060
[0003]         main:11 : free( 0x9d5060 )
[0004]         main:19 : free( 0x9d5060 )
[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.
```

v. Additional Test

realloc 의 경우 이전보다 block size 가 더 큰 경우에만 freed_total 값을 증가시키는지, size 가 0 인 경우에도 block 을 잘 할당하는지, invalid 한 pointer 값을 받았을 때 올바르게 처리하는지 여부를 확인하고자 별도의 test code 를 작성하였고 이를 이용하여 test 를 진행하였다.

```c
#include <stdlib.h>
int main(void)
{
  void *a = calloc(10, 0); //test for size = 0
  void *p = malloc(0); //test for size = 0
  void *ptr = malloc(100);
  ptr = realloc(ptr, 10); // free for 100 - 10 bytes
  void *ptr2 = realloc(NULL, 0); //realloc with NULL ptr
  int b = 2;
  ptr2 = realloc(&b, 10); //realloc with invalid pointer
  void *double_realloc = malloc(10);
  void *first_realloc = realloc(double_realloc, 1000);
  first_realloc = realloc(double_realloc, 4); //test for double-realloc
  return 0;
}
```

```
stu114@sp1:~/SP2019/linklab/handout/bonus$ make run testy
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c ../utils/memlist.c ca
llinfo.c -ldl -lunwind
[0001] Memory tracer started.
[0002]         main:1c : calloc( 10 , 0 ) = 0x127e060
[0003]         main:23 : malloc( 0 ) = 0x127e0d0
[0004]         main:2d : malloc( 100 ) = 0x127e140
[0005]         main:3a : realloc( 0x127e140 , 10 ) = 0x127e140
[0006]         main:43 : realloc( (nil) , 0 ) = 0x127e200
[0007]         main:5a : realloc( 0x7ffe154784e4 , 10 ) = 0x127e220
[0008]             *** ILLEGAL_FREE *** (ignoring)
[0009]         main:64 : malloc( 10 ) = 0x127e290
[0010]         main:74 : realloc( 0x127e290 , 1000 ) = 0x127e300
[0011]         main:81 : realloc( 0x127e290 , 4 ) = 0x127e290
[0012]             *** DOUBLE_FREE  *** (ignoring)
[0013]
[0014] Statistics
[0015]   allocated_total      1134
[0016]   allocated_avg        126
[0017]   freed_total          90
[0018]
[0019] Non-deallocated memory blocks
[0020]   block              size       ref cnt    caller
[0021]   0x127e060          0          1          main:1c
[0022]   0x127e0d0          0          1          main:23
[0023]   0x127e140          10         1          main:2d
[0024]   0x127e200          0          1          main:43
[0025]   0x127e220          10         1          main:5a
[0026]   0x127e290          4          1          main:64
[0027]   0x127e300          1000       1          main:74
[0028]
[0029] Memory tracer stopped.
```

우선 calloc, malloc, realloc 에 대해서 size 가 0 인 경우에 대해서도 block 을 잘 할당하는지 확인하였다. 아래 결과에서와 같이 제대로 block 을 할당하였으며 이 경우에도 free 를 해주지 않는 경우 non-deallocated memory block 에 나타남을 확인할 수 있었다. realloc 에 NULL 을 인자로 넘겨주는 경우에도 새로운 block 을 할당하는 사실도 test 를 통해 확인할 수 있다. ptr2 = realloc(&b, 10); 에서 invalid 한 pointer 를 realloc 에 넘겨주는 경우 illegal free error 를 출력하는 것도 확인할 수 있다. 이미 realloc 을 통해 free 된 block 을 또 한번 realloc 을 통해 free 하고자 하면 double free error 를 출력하는 것 또한 확인할 수 있다. illegal 한 free 를 요구하는 realloc 에서는 freed_total 값이 증가하지 않고, valid 한 realloc 중에서 새로 할당받은 block 의 size 가 원래 size 보다 더 작은 경우에만 차이값을 더하는 것 또한 freed_total 값이 100 -10 = 90 만큼 증가하는 것을 통해 확인하였다. 위의 결과에서 realloc 이 된 block 의 경우가 free 가 되지 않는 경우 caller 에 해당하는 부분이 처음 memory 를 할당받은 instruction 의 위치로 나오는 것은 alloc 함수에서 함수 이름과 해당 instruction offset 을 update 하지 않기 때문인데, alloc 함수를 조금만 수정하면 이 값 또한 마지막 realloc 이 불린 위치로 update 를 할 수 있으나 이번 lab 에서는 별도로 기존 utils/memlist.c 파일을 수정하지 않았다.

Conclusion

이번 linklab 을 통해 load/time interpositioning 에 대한 개념을 보다 정확히 익힐 수 있었다. 실제로 log 를 찍으며 tracing 을 함으로써 debugging 을 할 때 유용한 방법으로 사용될 수 있다는 것을 직접 느낄 수 있었다. 또한, 직접 구현을 하지는 않았지만, utils/memlist.c 의 function 들을 이해하면서 linked list 를 활용하여 어떻게 dynamic memory allocation 된 block 들을 tracing 할 수 있을지 알고리즘에 대해 생각해 볼 수 있었다. 또한 Part3 의 부분에서 <libunwind.h>의 library 를 이용하여 procedure 와 instruction 을 tracing 하는 과정에서 많은 것을 배울 수 있었다. 우선 해당 library 의 존재를 이번 lab 을 통해 처음 접하게 되었으며 unw_get_proc_name 과 같은 function 을 통해 tracing 하는 법을 익힐 수 있었다. 이를 사용하면서 혁신적인 debugging tool 로만 느껴지던 gdb 가 이와 같은 library 와 같은 원리로 동작함을 추측할 수 있었다.
처음 이 lab 과제를 하면서 memtrace.c 파일 이외에 utils directory 안에 있는 다양한 function 과 macro 를 이해하는 과정에서 시간이 좀 걸리긴 하였으나 실질적인 구현은 비교적 어렵지 않게 할 수 있었다. 다만 이번 lab 에 있어서 testcase1 에 대한 예시 답안만 제공되어 해당 test 에서 다루지 않는 corner case 혹은 realloc 함수에 대한 처리를 어떻게 해주어야 하는지 명확하지 않았던 점이 이번 과제에서 가장 어려웠던 점이라고 생각한다.