[System Programming] Linklab report

- Name :
- Student ID :

Dynamic Memory Management

> 이 랩 세션에서는 4개의 동적 메모리 관리 함수인 malloc, calloc, realloc, free 함수의 wrapper 함수를 만들고 그 수행과정과 결과에 대해서 log를 출력해 볼 것이다. 이 보고서에서는 PART1부터 PART3, 그리고 BONUS 파트까지 어떻게 구현을 했고, 구현 과정에서 느꼈던 점을 서술할 것이다.

PART1

파트1 에서는 동적 메모리 관리 함수들의 이름, 매개변수, 리턴값들을 출력하고, 프로그램이 종료되면 총 할당 메모리, 할당 메모리 평균값을 출력한다. 파트1에서는 아직 free된 총 메모리 바이트나 deallocate되지 않은 메모리 블록을 출력할 필요는 없다. 파트1에서 구현한 malloc, calloc, realloc, free함수는 이후 다른 파트에서도 재사용될 것이므로 여기에서 구현을 잘 해놓는 것이 중요하다.

우선 전역변수를 먼저 살펴보자.

```
static unsigned long n_malloc  = 0;
static unsigned long n_calloc  = 0;
static unsigned long n_realloc = 0;
static unsigned long n_allocb  = 0;
static unsigned long n_freeb   = 0;
static item *list = NULL;

static item *freedlist = NULL;
```

- n_malloc, n_calloc, n_realloc은 각 함수가 호출된 횟수를 의미한다. 이는 fini 부분에서 statistic 중에 allocated_avg를 계산할 때 사용이 될 것이다.
- n_allocb는 전체 할당된 메모리 사이즈이며, n_freeb는 free된 메모리 사이즈이다.
- item 타입으로 선언된 list 변수는 할당된 메모리 블록을 담는 linkedlist 포멧의 자료구조이다.
- 여기까지는 스켈레톤 코드에서 제공된 변수들인데 나는 deallocate된 메모리 블록을 tracing 하기 위해서 freedlist라는 리스트를 하나더 선언하였다. 새롭게 할당된 메모리 블록은 우선 list에 삽입될 것이고, list에서 deallocate된 메모리 블록은 freedlist로 삽입될 것이다. deallocate된 메모리 블록을 따로 보관하는 이유는 비정상적인 free나 realloc이 발생하였을 때 이것이 illegal free인가 double free인가를 구분하기 위함이다.

> **과제를 하면서 memlist.c 파일의 dealloc함수에 일부 수정을 하였다. 이에 대한 내용은 part2를 설명하면서 구체적으로 설명할 것이다. 일단은 dealloc함수가 일반적인 linked list의 remove함수와 동일한 기능을 하도록 변경했다고만 언급하고 넘어가겠다**

이제 init 부분을 살펴보도록 하겠다.

```c
__attribute__((constructor))
void init(void)
{
  char *error;

  LOG_START();

  list = new_list();
  freedlist = new_list();
  // ...
  mallocp = dlsym(RTLD_NEXT, "malloc");
  if((error = dlerror())!=NULL)
  {
    fputs(error, stderr);
    exit(1);
  }

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


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

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

사실 init, fini와 같은 함수를 써보는 건 이번이 처음이라서 이 부분이 어떤 역할을 하는지 이해하기가 어려웠다. 조교님께서는 init을 간단히 설명하자면 java나 c++에서의 constructor 역할을 하는 부분이라고 생각하면 된다하셔서, constructor에서 그러하듯이 여기 부분에서 initialize 가능한 변수들은 초기화를 하였다. dlsym 함수 같은 경우 malloc, calloc, realloc, free 함수 내에서 사용해도 되지만 init 부분을 생성자라고 간주하여 여기에서 사용을 하였다.

그 다음은 fini 부분이다.

```c
__attribute__((destructor))
void fini(void)
{
  // ...

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

  LOG_STOP();

  free_list(list);
  free_list(freedlist);
}
```

이 부분은 shared library가 unload될 때 딱 한번 실행되는 부분으로, 이번 과제에서는 최종 통계 로그를 출력하고, list들을 free해주는 용도로 사용이 된다. 파트1에서는 freed total은 고려하지 않으므로 항상 0으로 출력될 수 있도록 LOG_STATISTICS 매크로의 마지막 인자로 0을 전달한다.

이제 동적 메모리 관리 함수들을 하나하나 살펴보자.

1. malloc

```c
void *malloc(size_t size)
{
  void *ptr = mallocp(size);
  n_allocb += size;
  n_malloc++;
  alloc(list, ptr, size);
  LOG_MALLOC(size, ptr);

  return ptr;
}
```

앞서 언급했지만 로드된 dynamic shared object를 조작할 수 있도록 하는 dlsym 함수는 init에서 구현했기 때문에 malloc 함수 내에서는 malloc의 동작만 재정의하면 된다. alloc함수를 통해 list에 할당된 메모리 블록을 삽입해주고, LOG_MALLOC을 통해서 로그를 출력한다.

2. calloc

```c
void *calloc(size_t nmemb, size_t size)
{
  void *ptr = callocp(nmemb, size);
  n_allocb += nmemb * size;
  n_calloc++;
  alloc(list, ptr, nmemb*size);
  LOG_CALLOC(nmemb, size, ptr);

  return ptr;
}
```

구현방식은 malloc과 거의 차이가 없고 original calloc의 기능처럼 nmeb * size만큼의 메모리 블록을 할당시킨다.

3. realloc

```c
void *realloc(void *ptr, size_t size)
{
  n_allocb += size;
  n_realloc++;

  if(find(list, ptr) == NULL)
  {
    void* old_ptr = ptr;
    ptr = reallocp(NULL, size);
    // According to C Standard, realloc(NULL, size) is the same as malloc(size)
    alloc(list, ptr, size);
    LOG_REALLOC(old_ptr, size, ptr);

    if(find(freedlist, ptr) == NULL)
      LOG_ILL_FREE();
    else
      LOG_DOUBLE_FREE();

    return ptr;
  }
  else
  {
    void* old_ptr = ptr;

    unsigned long old_size = find(list, old_ptr)->size;

    if(old_size >= size)         // if new allocation space is smaller than before
    {
      n_freeb += old_size - size;
    }

    dealloc(list, old_ptr);                // dealloc original ptr
    alloc(freedlist, old_ptr, old_size);  // put it into freedlist

    ptr = reallocp(old_ptr, size);
    alloc(list, ptr, size);            // alloc new ptr
    LOG_REALLOC(old_ptr, size, ptr);

    return ptr;
  }
}
```

realloc 함수 부분은 이번 과제에서 가장 구현하면서 고생을 많이 했다. 코딩을 하는 것은 큰 어려움이 없었는데 etl을 수시로 체크하면서 어떤 식으로 구현을 하고 예외처리를 해야하는지 신경쓰는데에 많은 에너지가 소모된 것같다. etl 공지사항 4번글("과제 1번 realloc 처리에 관해")에서는 정상적인 realloc 상황과 비정상적인 realloc 상황을 구분한다.

1. 정상적인 realloc
   - ptr이 가리키던 allocated block의 size = old_size
   - if(old_size – size) >= 0, n_freeb += old_size – size.
   - if(old_size - size) < 0, n_freeb = n_freeb (바꾸지 않음)
2. 비정상적인 realloc (ptr이 참조하는 블록이 allocate되지 않은 상황)
   - LOG_DOUBLE_FREE / LOG_ILL_FREE로 오류 로그 출력
   - return original_realloc(NULL, size)

위의 코드를 살펴보면 첫번째 조건문 `if(find(list, ptr) == NULL)`의 경우가 할당된 메모리를 보관하는 list 내에 해당 ptr이 들어있지 않은 비정상적인 **realloc의 상황**이다. 이 상황에서는 다음과 같이 처리 하였다.

- `void* old_ptr = ptr`
  - LOG 출력시 realloc 함수 호출이전의 ptr주소가 포함되어야 하기 때문에 이 주소는 reallocp를 하기 전

에 미리 보관을 해주어야한다.

- `prt = reallocp(NULL, size)`
    - 공지사항의 규정에 따라 original realloc에 NULL, size를 매개변수로 전달한다. 구글링을 해보니 C Standard에 따르면 **realloc(NULL, size)는 malloc(size)와 동일한 작동을 한다고** 한다.
- `alloc(list, ptr, size)`
    - 비정상적인 realloc의 경우 메모리 블록이 할당된 것으로 간주해야하는가 고민을 했지만, C standard 에서는 이를 malloc(size)와 동일하게 취급한다고 하였으니 이 경우에도 alloc함수를 호출하여 size만큼 메모리 할당을 한다.
- LOG_REALLOC(old_ptr, size, ptr)
    - 비정상적인 realloc에서도 LOG_REALLOC은 변함없이 호출되어야한다.
- if(find(freedlist, ptr) == NULL) LOG_ILL_FREE()
    - 앞에서 전역변수들에 대해 설명을 하면서 언급을 했던 freedlist에는 list에서 deallocate된 블록들이 담겨있다. 만약 freedlist에서 ptr을 find하였는데 NULL이 리턴된다면(freedlist에 ptr이 담겨있지 않다면. 즉, list에 해당블록이 할당된 적이 없었다면) 두 번 할당된 것이 아니므로 LOG_ILL_FREE를 호출한다.
- else LOG_DOUBLE_FREE()
    - 그 외의 경우엔 LOG_DOUBLE_FREE를 호출한다.
- return ptr
    - 비정상 realloc에 대한 처리가 끝났으므로 ptr을 리턴한다.

반면, **정상적인 realloc의 경우엔 다음과 같이 처리한다.**

- `void* old_ptr = ptr`
    - LOG 출력시 realloc 함수 호출이전의 ptr주소가 포함되어야 하기 때문에 이 주소는 reallocp를 하기 전에 미리 보관을 해주어야한다.
- `unsigned long old_size = find(list, old_ptr)->size`
    - 공지사항의 realloc 규정에 따르면, old_size가 새로 할당되는 size보다 큰 경우에만 n_freeb를 old_size - size만큼 증가시킨다. 이를 위해서 list에 있는 old_ptr 블록의 size를 old_size 변수에 따로 저장해둔다.
- `if(old_size >= size) n_freeb += old_size - size`
    - 공지사항의 realloc 규정에 따라 old_size가 새롭게 할당되는 size보다 큰 경우 n_freeb를 old_size-size 만큼 증가시킨다.
    - 과제를 하면서 이 부분을 디버깅하는데 시간이 좀 걸렸다. 공지사항에서는 `if(old_size – size >= 0)`라고 언급을 해놔서 처음엔 조건문을 `if(old_size >= size)`가 아니라 `if(old_size – size >= 0)`로 썼는데 알 수 없는 에러가 발생했다. 그 이유를 추적해보니 old_size와 size 모두 변수의 타입이 unsinged라서 old_size - size를 하면 old_size가 더 작은 경우에도 양수(매우 큰 양수)가 나오는 상황이 발생했고, 이 조건문은 항상 참이 되었다. 컴퓨터구조 수업에서 unsigned 계산시에는 주의를 기울여야한다고 했던 내용을 복습하는 순간이었다.

- `dealloc(list, old_ptr)`
  - **기존에 할당되어있던** old_ptr**을** dealloc **해준다**.
- `alloc(freedlist, old_ptr, old_size)`
  - **앞에서** dealloc**한** old_ptr **블록을** freedlist**에 삽입해준다**.
- `ptr = reallocp(old_ptr, size)`
  - original realloc **함수를 통해 재할당을 진행한다**.
- `alloc(list, ptr, size)`
  - **새롭게 재할당된** ptr **메모리 블록을** list**에** allocate**한다**.
- `LOG_REALLOC(old_ptr, size, ptr)`
  - **매크로를 통해 로그를 남긴다**.
- `return ptr`
  - ptr**을 리턴한다**.

4. free

```c
void free(void *ptr)
{
  if(!ptr)
    return;

  unsigned long freeb = find(list, ptr)->size;
  n_freeb += freeb;
  dealloc(list, ptr);
  alloc(freedlist, ptr, freeb);
  LOG_FREE(ptr);

  freep(ptr);
}
```

memtrace.c**에서 마지막으로 살펴볼** free **함수이다**.

- freeb**라는 로컬 변수를 따로 선언하여** free**할** ptr**의** size**를 저장한다**.
- **전체** freed byte**를 저장하는** n_freeb **전역변수에** freeb**의 값을 더해준다**.
- list**에서** ptr **메모리 블록을** deallocate **해준다**.
- list**에서** deallocate**된 메모리 블록은** freedlist**에 삽입된다**.
- LOG_FREE**를 통해 로그를 남긴다**.
- original free**함수를 통해** ptr**을** free **해준다**.

PART2

파트2**에서 파트**1**에서** 0**으로만 출력하던** freed_total**을 제대로 계산하여 출력하고**, Non-deallocated memory blocks**에 대해서도 로그를 남기도록 구현해야한다**.

앞에서 파트1을 설명하면서 memtrace.c 파일에 대해서는 충분히 자세히 설명을 하였으므로 이제부터는 변경된 부분에 한해서만 설명을 하도록 하겠다. 우선 memtrace.c를 살펴보기 전에 utils 디렉토리에 있는 memlist.c 파일의 dealloc함수를 먼저 살펴보자. 다음은 내가 변경을 하기 전에 스켈레톤 코드에 들어있던 dealloc 함수의 내용이다.

```
 1 item *dealloc(item *list, void *ptr)
 2 {
 3   item *prev, *cur, *i;
 4
 5   if (list == NULL) return NULL;
 6
 7   // find block
 8   prev = list; cur = list->next;
 9   while ((cur != NULL) && (cur->ptr != ptr)) {
10     prev = cur; cur = cur->next;
11   }
12
13   // decrement reference count if found
14   if (cur != NULL) cur->cnt--;
15
16   /**** This part will be changed! ****/
17
18   return cur;
19 }
```

스켈레톤 코드로 제공된 memlist 라이브러리의 item 타입 구조체는 linked list와 비슷한 자료구조를 가지고 있다. 그래서 처음엔 alloc 함수는 linked list의 insert와 비슷하고, dealloc 함수는 remove와 비슷할 것이라고 생각을 했는데 dealloc 함수의 작동 방식이 linked list의 것과는 차이가 있었다. 스켈레톤 코드 상에서는 dealloc이 호출되었을 때 list 상의 node를 삭제하는 것이 아니라 그냥 cnt만 줄이는 식으로 구현이 되어있었다. 스켈레톤 코드를 그대로 써서 memtrace.c를 완성할 수도 있겠지만 나는 보다 익숙한 linked list와 같은 방식으로 작동시키기 위해 다음과 같이 코드를 변경하였다. 16, 17번 라인이 새롭게 추가된 부분이다.

```
 1 item *dealloc(item *list, void *ptr)
 2 {
 3   item *prev, *cur, *i;
 4
 5   if (list == NULL) return NULL;
 6
 7   // find block
 8   prev = list; cur = list->next;
 9   while ((cur != NULL) && (cur->ptr != ptr)) {
10     prev = cur; cur = cur->next;
11   }
12
13   // decrement reference count if found
14   if (cur != NULL) {
15     cur->cnt--;
16     prev->next = cur->next;
17     freep(cur);
18   }
19
20   return cur;
21 }
```

이와 같이 구현을 하면 dealloc 함수 호출시 list상에서 ptr에 해당하는 노드를 삭제하게 된다.

그럼 이제 다시 memtrace.c를 살펴보자. 파트1과 비교했을 때 달라진 부분은 fini 함수 부분이다.

```c
__attribute__((destructor))
void fini(void)
{
  // ...

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

  if(list->next != NULL)
    LOG_NONFREED_START();

  item* temp = list->next;
  while(temp != NULL)
  {
    LOG_BLOCK(temp->ptr, temp->size, temp->cnt, temp->fname, temp->ofs);
    temp = temp->next;
  }

  LOG_STOP();

  free_list(list);
  free_list(freedlist);
}
```

- 우선 freed_total을 출력해야하기 때문에 LOG_STATISTICS 매크로의 마지막 인자로 0대신 전역변수인 n_freeb가 전달이 된다. 이제는 realloc과 free함수를 통해서 free 처리된 메모리 사이즈가 로그에 남을 것이다.
- 파트2에서부터는 Non-deallocated memory blocks를 출력해야한다.
  - 만약 list상에 담겨있는 원소가 없는(non-deallocated memory block이 하나도 없는) 상태라면 LOG_NONFREED_START()를 출력하지 않고, 그 외의 경우엔 이를 출력한다.
  - item 타입의 temp 구조체를 선언하여 list의 head 부분을 가리키도록 한다. 그리고 while문 내에서 list가 tail에 도달할 때까지 담겨있는 메모리 블록을 출력한다. fini가 호출되는 시점에 list 내에 남아있는 원소는 deallocate되지 않은 메모리 블록을 의미하므로 이들을 모두 출력하는 것이다.
- LOG_STOP을 출력하고, list와 freedlist를 둘다 destruct 시켜준다.

PART3

파트3에서는 로그를 남길 때 동적 메모리 관리 함수를 호출한 인스트럭션의 함수 이름과 오프셋까지 출력이 되도록 한다. 이를 위해서는 callinfo.c 함수를 작성하여 backtracing을 하도록 해야한다. libunwind.h 라이브러리가 제공이 되었기 때문에 이를 통해서 큰 어려움 없이 backtracing을 할 수 있었다.

```c
#include <stdlib.h>
#define UNW_LOCAL_ONLY
#include <libunwind.h>
#include <string.h>

int get_callinfo(char *fname, size_t fnlen, unsigned long long *ofs)
{
  unw_context_t context;
  unw_cursor_t cursor;
  unw_word_t off;
  char proc_name[256];

  if(unw_getcontext(&context))
    return -1;

  if(unw_init_local(&cursor, &context))
    return -1;

  unw_step(&cursor);
  unw_step(&cursor);
  unw_step(&cursor);
  unw_get_proc_name(&cursor, proc_name, 256, &off);


  *ofs = off-5;
  strncpy(fname, proc_name, fnlen);
  return 0;
}
```

- `#define UNW_LOCAL_ONLY` 매크로 부분을 선언하지 않으면 에러가 발생한다.
- unw_step을 3번 해주는 이유는 동적 할당 함수들이 호출되는 곳을 거슬러 올라가려면 3번의 step을 거쳐야 하기 때문이다. 첫번째로는 get_callinfo 함수로 거슬러 올라가고, 그 다음은 malloc, calloc, realloc, free 함수, 그리고 그 다음에서야 앞의 동적 함수들을 호출한 함수로 거슬러 올라가진다.
- ofs는 함수의 오프셋을 계산할 때 사용이 되는데 unw_get_proc_name을 통해서 얻게된 off에 5를 빼주어야 정확한 위치가 계산이 된다. objdump를 해보면 callq 인스트럭션의 사이즈가 4 bytes*5이기 때문에 5를 빼준다.
- 마지막으로 fname에는 unw_get_proc_name 함수를 통해 얻은 proc_name(caller 함수명)을 string copy 해준다.

이렇게 구현이 완료되면 이제 memlog 헤더파일에서 LOG를 위한 매크로를 호출할 때 get_callinfo함수가 완전한 기능을 함에 따라 정확한 caller 함수 이름과 그 오프셋이 출력된다.

BONUS

보너스 파트는 test 디렉토리에 있는 테스트케이스 중 하나인 test4와 같이 비정상적인 free가 발생하였을 때, 에러가 발생하여 프로그램이 종료되는 것이 아니라 c++이나 java의 try, catch처럼 일종의 예외처리가 되도록하여 해당 명령을 무시하고 illegal free, double free와 같은 로그를 출력하도록 해야한다.

이를 위해서는 memtrace.c 파일의 free 함수를 수정해야한다. 수정된 free 함수는 다음과 같다.

```c
void free(void *ptr)
{
  if(!ptr)
    return;

  if(find(list, ptr) == NULL)
  {
    LOG_FREE(ptr);
    if(find(freedlist, ptr) == NULL)
    {
      LOG_ILL_FREE();
      return;
    }
    else
    {
      LOG_DOUBLE_FREE();
      return;
    }
  }
  else
  {
    unsigned long freeb = find(list, ptr)->size;
    n_freeb += freeb;
    dealloc(list, ptr);
    alloc(freedlist, ptr, freeb);
    LOG_FREE(ptr);

    freep(ptr);
  }
}
```

1. 비정상적인 free가 발생한 경우(기존의 오류 상황)

- `if(find(list, ptr) == NULL)`인 경우가 이에 해당한다. 즉, list에서 ptr이 존재하지 않는 상황이다. 할당되지 않는 메모리 블록을 free하는 상황이므로 에러로그를 출력해줘야한다.
- 우선 과제 스펙에 따라 LOG_FREE를 먼저 출력을 한다.
- 비정상 free에는 2가지 경우가 있는데 이 경우들을 나눠서 처리해주어야한다.
  - **Illegal Free인 경우**
    `if(find(freedlist, ptr) == NULL)`인 경우, 즉 지금까지 한번도 할당된 적이 없는 메모리 블록을 deallocate하려는 경우 → LOG_ILL_FREE를 호출하고 return 한다.
  - **Double Free인 경우**
    freedlist에 ptr이 존재하는 경우이다. 즉, 현재 list에는 allocate되지는 않은 상태이지만 과거에 list에 allocate가 되었다가 deallocate가 된 경우이다. 이는 동일한 메모리 블록이 2번 free되었음을 의미한다.
    → LOG_DOUBLE_FREE를 호출하고 return 한다.
- 정상적인 free의 경우에는 파트1~파트3에서의 free함수와 완전히 동일하게 동작한다.

느낀점