Lab 1: Linklib

공과대학 컴퓨터공학부
2020-14378
윤교준 / Gyojun Youn
youngyojun@snu.ac.kr

Part 1

구현

먼저, `dlsym` 함수를 통해 공유 오브젝트 중에서 표준 함수들 `malloc`, `calloc`, `realloc`, `free` 심볼이 메모리에 적재된 주소를 가져와야 한다. `RTLD_NEXT` 핸들을 사용하면 라이브러리에서 두 번째로 찾은 함수의 주소를 받아올 수 있기 때문에 정당하다.

```
//
// init - this function is called once when the shared library is loaded
//
__attribute__((constructor))
void init(void)
{
  char *error;

  dlerror();

  LOG_START();

  if (!mallocp)
  {
    mallocp = dlsym(RTLD_NEXT, "malloc");
    if ((error = dlerror()) || !mallocp)
    {
      fprintf(stderr, "Error getting symbol 'malloc': %s\n", error);
      exit(EXIT_FAILURE);
    }
  }

  if (!callocp)
  {
    callocp = dlsym(RTLD_NEXT, "calloc");
    if ((error = dlerror()) || !callocp)
    {
      fprintf(stderr, "Error getting symbol 'calloc': %s\n", error);
      exit(EXIT_FAILURE);
    }
  }
```

```
  if (!reallocp)
  {
    reallocp = dlsym(RTLD_NEXT, "realloc");
    if ((error = dlerror()) || !reallocp)
    {
      fprintf(stderr, "Error getting symbol 'realloc': %s\n", error);
      exit(EXIT_FAILURE);
    }
  }

  if (!freep)
  {
    freep = dlsym(RTLD_NEXT, "free");
    if ((error = dlerror()) || !freep)
    {
      fprintf(stderr, "Error getting symbol 'free': %s\n", error);
      exit(EXIT_FAILURE);
    }
  }
}
```

`dlsym` 함수의 반환값이 `NULL` 인지, 그리고 `dlerror` 함수의 반환값이 non-`NULL` 인지 여부를 확인함으로써, 공유 라이브러리를 load하는데 오류가 있었는지 확인하였다. 함수 처음에 `dlerror`의 호출은 기존에 남아있던 error message를 flush하기 위하여 필요하다.

이제, 메모리 관련 표준 함수를 `mallocp`, `callocp`, `reallocp`, `freep`의 이름으로 사용할 수 있다.

`void* malloc(size_t size)`는 다음과 같이 구현하였다. Total allocated size와 allocation count를 관리해야 한다. 관련 log는 `utils/memlog.h`에 정의된 `LOG_MALLOC` define 지시문을 사용하였다.

```
void* malloc(size_t size)
{
  void *ptr = mallocp(size);

  LOG_MALLOC(size, ptr);

  n_malloc += size;
  n_allocb++;

  return ptr;
}
```

같은 방법으로 `void* calloc(size_t nmemb, size_t size)`과 `void* realloc(void* ptr, size_t size)`을 구현하였다. Reallocation의 경우, 항상 해제 후 할당 받는다고 가정하고 통계량을 관리해야 함에 유의하였다.

```
void* calloc(size_t nmemb, size_t size)
{
  void *ptr = callocp(nmemb, size);
```

```
  LOG_CALLOC(nmemb, size, ptr);

  n_calloc += size;
  n_allocb++;

  return ptr;
}

void* realloc(void* ptr, size_t size)
{
  void *nptr = reallocp(ptr, size);

  LOG_REALLOC(ptr, size, nptr);

  n_realloc += size;
  n_allocb++;

  return nptr;
}
```

`void free(void* ptr)`의 구현은 직관적이고 명확하다.

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

`fini` 함수에서는 관련 통계량을 출력해야 한다. Allocated size를 관리하는 세 변수 `n_malloc`, `n_calloc`, `n_realloc`을 사용하면 충분하다.

```
//
// fini - this function is called once when the shared library is unloaded
//
__attribute__((destructor))
void fini(void)
{
  unsigned long alloc_tot = n_malloc + n_calloc + n_realloc;
  unsigned long alloc_avg = n_allocb ? alloc_tot / n_allocb : 0;

  LOG_STATISTICS(alloc_tot, alloc_avg, n_freeb);

  LOG_STOP();
}
```

실행 결과

Test 1

먼저, `test1.c`의 코드 내용은 다음과 같다.

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

실습 수업용 서버에서 실행한 결과는 아래와 같다. 이후 모든 실행 결과는 같은 환경에서 얻었다.

```
stu70@sp01:~/Labs/sp-linklab/linklab/part1$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 1024 ) = 0x559fa40a32a0
[0003]           malloc( 32 ) = 0x559fa40a36b0
[0004]           malloc( 1 ) = 0x559fa40a36e0
[0005]           free( 0x559fa40a36e0 )
[0006]           free( 0x559fa40a36b0 )
[0007]
[0008] Statistics
[0009]   allocated_total      1057
[0010]   allocated_avg        352
[0011]   freed_total          0
[0012]
[0013] Memory tracer stopped.
```

Total freed size는 관리하지 않았기 때문에, 통계량의 `freed_total`이 0임에 유의하라.

Test 2

아래는 `test2.c`의 코드 내용이다.

```
a = malloc(1024);
free(a);
```

```
stu70@sp01:~/Labs/sp-linklab/linklab/part1$ make run test2
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 1024 ) = 0x55bcca2372a0
[0003]           free( 0x55bcca2372a0 )
[0004]
[0005] Statistics
[0006]   allocated_total      1024
[0007]   allocated_avg        1024
[0008]   freed_total          0
[0009]
[0010] Memory tracer stopped.
```

Test 3

```c
void *a[10];
int i;

for (i=0; i<10; i++) {
  size_t s = rand() % (1<<16);
  a[i] = rand() % 2 ? malloc(s) : calloc(1, s);
}

for (i=10; i>0; i--) free(a[i-1]);
```

```
stu70@sp01:~/Labs/sp-linklab/linklab/part1$ make run test3
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 18317 ) = 0x5601121182a0
[0003]           calloc( 1 , 47914 ) = 0x56011211ca40
[0004]           calloc( 1 , 51385 ) = 0x560112128580
[0005]           malloc( 38071 ) = 0x560112134e50
[0006]           calloc( 1 , 42510 ) = 0x56011213e310
[0007]           calloc( 1 , 19424 ) = 0x560112148930
[0008]           malloc( 26331 ) = 0x56011214d520
[0009]           calloc( 1 , 4368 ) = 0x560112153c10
[0010]           calloc( 1 , 48632 ) = 0x560112154d30
[0011]           calloc( 1 , 45617 ) = 0x560112160b30
[0012]           free( 0x560112160b30 )
[0013]           free( 0x560112154d30 )
[0014]           free( 0x560112153c10 )
[0015]           free( 0x56011214d520 )
[0016]           free( 0x560112148930 )
[0017]           free( 0x56011213e310 )
[0018]           free( 0x560112134e50 )
[0019]           free( 0x560112128580 )
```

```
[0020]           free( 0x56011211ca40 )
[0021]           free( 0x5601121182a0 )
[0022]
[0023] Statistics
[0024]   allocated_total      342569
[0025]   allocated_avg        34256
[0026]   freed_total          0
[0027]
[0028] Memory tracer stopped.
```

이 결과는 매 실행마다 다르다. 할당받은 포인터의 주소와 해제되는 포인터의 주소의 순서가 서로 역순임을 확인할 수 있다. 또한, 통계량의 `allocated_avg`의 값이 `floor( allocated_total / 10 )`과 같음을 볼 수 있다.

Part 2

구현

Part 1의 코드에 이어서 total freed size와 non-deallocated blocks을 관리해야 한다. 후자를 관리하기 위하여 (block address, size) 쌍의 정보를 담고 있는 linked-list 구조의 `utils/memlist.c`의 `item`을 사용하였다.

먼저, `new_list` 함수를 호출하여 자료구조 `list`를 얻었다.

```
//
// init - this function is called once when the shared library is loaded
//
__attribute__((constructor))
void init(void)
{
  char *error;

  dlerror();

  LOG_START();

  // initialize a new list to keep track of all memory (de-)allocations
  // (not needed for part 1)
  list = new_list();

  /// 후략
}
```

`malloc`, `calloc` 함수는 할당받은 메모리 주소와 크기를 `list`에 `alloc` 함수를 통하여 기록해야 한다.

```
void* malloc(size_t size)
{
  void *ptr = mallocp(size);
```

```
  alloc(list, ptr, size);
  LOG_MALLOC(size, ptr);

  /// 후략
}

void* calloc(size_t nmemb, size_t size)
{
  void *ptr = callocp(nmemb, size);

  alloc(list, ptr, size);
  LOG_CALLOC(nmemb, size, ptr);

  /// 후략
}
```

`realloc` 함수는 조금 까다롭다. 기존의 `ptr` 메모리의 정보를 `list`에서 제거하고, 새로 할당받은 정보를 삽입해야 한다. 또한, 이 과정에서 기존 `ptr`이 가리키는 메모리 영역의 크기를 알아내어, total freed size에 반영해야 한다. 후자의 작업은 `utils/memlist.c`의 `find` 함수로 쉽게 구현할 수 있다.

```
void* realloc(void* ptr, size_t size)
{
  void *nptr = reallocp(ptr, size);

  LOG_REALLOC(ptr, size, nptr);

  n_freeb += find(list, ptr)->size;
  n_realloc += size;
  n_allocb++;

  dealloc(list, ptr);
  alloc(list, nptr, size);

  return nptr;
}
```

`find(list, ptr)`, `dealloc(list, ptr)`, `alloc(list, nptr, size)`의 호출 순서에 유의하라.

마찬가지로, `free` 함수는 직관적이다.

```
void free(void* ptr)
{
  freep(ptr);

  n_freeb += find(list, ptr)->size;

  dealloc(list, ptr);
  LOG_FREE(ptr);
}
```

이제, `fini` 함수에서 non-deallocated block을 찾고 이를 보고하는 코드를 작성해야 한다. Linked-list `list`의 entry를 순회하면서, reference count가 양수인 (address, size) 쌍이 있는지 검사하고, 만일 있다면 이를 보고하면 충분하다. 할당되면 reference count는 1 증가하고, 해제되면 1 감소하기 때문에, 이러한 구현은 정당하다.

```
//
// fini - this function is called once when the shared library is unloaded
//
__attribute__((destructor))
void fini(void)
{
  unsigned long alloc_tot = n_malloc + n_calloc + n_realloc;
  unsigned long alloc_avg = n_allocb ? alloc_tot / n_allocb : 0;
  int nonfreed_flag = 0;

  LOG_STATISTICS(alloc_tot, alloc_avg, n_freeb);

  for (item *i = list; i; i = i->next)
  {
    if (0 < i->cnt)
    {
      if (!nonfreed_flag)
      {
        nonfreed_flag = 1;
        LOG_NONFREED_START();
      }
      LOG_BLOCK(i->ptr, i->size, i->cnt);
    }
  }

  LOG_STOP();

  // free list (not needed for part 1)
  free_list(list);
  list = NULL;
}
```

`list`를 해제한 후, 이제는 invalid한 주소를 `NULL`로 덮어씌우는 과정을 잊어서는 안된다.

실행 결과

Part 1

```
stu70@sp01:~/Labs/sp-linklab/linklab/part2$ make run test1
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 1024 ) = 0x5638f777b2d0
[0003]           malloc( 32 ) = 0x5638f777b710
[0004]           malloc( 1 ) = 0x5638f777b770
[0005]           free( 0x5638f777b770 )
[0006]           free( 0x5638f777b710 )
[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
[0015]   0x5638f777b2d0     1024       1
[0016]
[0017] Memory tracer stopped.
```

통계량의 `freed_total`이 이제 0이 아닌 33으로 잘 나온다. 또한, 1024 bytes의 `0x5638f777b2d0` addressed block이 해제되지 않았는데, 이것이 잘 보고되었다.

Part 2

```
stu70@sp01:~/Labs/sp-linklab/linklab/part2$ make run test2
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 1024 ) = 0x55b7c12bf2d0
[0003]           free( 0x55b7c12bf2d0 )
[0004]
[0005] Statistics
[0006]   allocated_total      1024
[0007]   allocated_avg        1024
[0008]   freed_total          1024
[0009]
[0010] Memory tracer stopped.
```

이 경우에는 할당-해제가 모두 짝을 이루어 잘 이루어졌기 때문에, non-deallocated memory blocks의 보고가 없다.

Part 3

```
stu70@sp01:~/Labs/sp-linklab/linklab/part2$ make run test3
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           calloc( 1 , 18621 ) = 0x55cbd7a172d0
[0003]           malloc( 30013 ) = 0x55cbd7a1bbd0
[0004]           calloc( 1 , 63738 ) = 0x55cbd7a23150
[0005]           malloc( 47016 ) = 0x55cbd7a32a90
[0006]           calloc( 1 , 52865 ) = 0x55cbd7a3e270
[0007]           malloc( 46970 ) = 0x55cbd7a4b130
[0008]           calloc( 1 , 13589 ) = 0x55cbd7a568f0
[0009]           malloc( 33016 ) = 0x55cbd7a59e40
[0010]           malloc( 41264 ) = 0x55cbd7a61f70
[0011]           calloc( 1 , 57486 ) = 0x55cbd7a6c0e0
[0012]           free( 0x55cbd7a6c0e0 )
[0013]           free( 0x55cbd7a61f70 )
[0014]           free( 0x55cbd7a59e40 )
[0015]           free( 0x55cbd7a568f0 )
[0016]           free( 0x55cbd7a4b130 )
[0017]           free( 0x55cbd7a3e270 )
[0018]           free( 0x55cbd7a32a90 )
[0019]           free( 0x55cbd7a23150 )
[0020]           free( 0x55cbd7a1bbd0 )
[0021]           free( 0x55cbd7a172d0 )
[0022]
[0023] Statistics
[0024]   allocated_total      404578
[0025]   allocated_avg        40457
[0026]   freed_total          404578
[0027]
[0028] Memory tracer stopped.
```

Part 3

구현

Part 2에 이어, illegal free와 double free를 탐지하고, 이러한 잘못된 free 명령에 대하여 오류 처리를 수행해야 한다.

함수 `void free(void* ptr)`의 구현만 고치면 충분하다. 먼저, `ptr`가 `list`의 entries 중 하나에 속하지 않는다면, 이는 `malloc` 등의 함수로 할당받은 주소가 아니다. 즉, 이 경우는 illegal free에 해당한다. 만일, 그 entry의 reference count가 양수가 아니라면, 이는 이미 해제된 메모리 영역을 다시 free하는 작업이기 때문에 double free에 해당한다.

이 두 경우에 해당한다면, 대응되는 오류 메시지를 출력한 후, 어떠한 추가 작업 없이 return하였다.

```
void free(void* ptr)
{
```

```
  item* i = find(list, ptr);

  LOG_FREE(ptr);

  if (!i)
  {
    LOG_ILL_FREE();
    return;
  }

  if (!i->cnt)
  {
    LOG_DOUBLE_FREE();
    return;
  }

  dealloc(list, ptr);
  n_freeb += i->size;

  freep(ptr);
}
```

`!i->cnt`에서 연산자 `->`가 `!`보다 우선순위가 높음에 유의하라.

실행 결과

Test 1부터 Test 3은 이전과 동일하게 의도대로 작동하였다.

Test 4

`test4.c`의 코드 내용은 다음과 같다.

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

`a`의 double free가 수행되고, 그 직후 임의의 주소에 대한 illegal free가 수행된다.

```
stu70@sp01:~/Labs/sp-linklab/linklab/part3$ make run test4
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 1024 ) = 0x55de1c6822d0
[0003]           free( 0x55de1c6822d0 )
[0004]           free( 0x55de1c6822d0 )
[0005]     *** DOUBLE_FREE  *** (ignoring)
[0006]           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.
```

의도대로 double free와 illegal free 관련 에러 메시지를 출력함을 확인할 수 있다. 또한, 이러한 오류 속에서도 `list` entries의 reference count가 잘 관리됨을 간접적으로 알 수 있다.

Test 5

`test5.c`의 코드 내용은 다음과 같다.

```
a = malloc(10);
a = realloc(a, 100);
a = realloc(a, 1000);
a = realloc(a, 10000);
a = realloc(a, 100000);
free(a);
```

여러 번의 reallocation 수행 후 free하는, 메모리적으로 안전한 코드다.

```
stu70@sp01:~/Labs/sp-linklab/linklab/part3$ make run test5
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 10 ) = 0x5569c06dc2d0
[0003]           realloc( 0x5569c06dc2d0 , 100 ) = 0x5569c06dc320
[0004]           realloc( 0x5569c06dc320 , 1000 ) = 0x5569c06dc3c0
[0005]           realloc( 0x5569c06dc3c0 , 10000 ) = 0x5569c06dc7e0
[0006]           realloc( 0x5569c06dc7e0 , 100000 ) = 0x5569c06def30
[0007]           free( 0x5569c06def30 )
[0008]
[0009] Statistics
[0010]   allocated_total      111110
[0011]   allocated_avg        22222
[0012]   freed_total          111110
[0013]
[0014] Memory tracer stopped.
```

예상과 같이, 어떠한 에러도 발생하지 않았으며, 할당-해제 또한 잘 이루어졌음을 확인할 수 있다.

Additional Tests

`test5.c`의 코드를 변형하여, 추가적인 테스트 코드 `test7.c`를 작성하였다.

```
#include <stdlib.h>

int main(void)
{
  void *a;

  a = malloc(10);
  a = realloc(a, 100);
  a = realloc(a, 1000);
  a = realloc(a, 10000);
  a = realloc(a, 100000);
  free(a);
  free(a);

  a = malloc(100000);
  free(a);
  free(a);
  free(a+4);

  return 0;
}
```

OS의 메모리 관리 방법을 생각하면, `a = realloc(a, 100000);`에서 할당 받은 주소는, 해제 후 `a = malloc(100000);`에서 (아주 높은 확률로) 동일한 주소를 할당 받으리라 예상할 수 있다. double free 오류 과정에서도 `list` entries의 reference count가 잘 관리되는지 확인하기 위한 테스트 코드다. 만일, reference count가 음수로 내려간다면, `a = malloc(100000);` 직후의 `free(a);`를 오류로 판단할 것이다.

```
stu70@sp01:~/Labs/sp-linklab/linklab/part3$ make run test7
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 10 ) = 0x5652318c22d0
[0003]           realloc( 0x5652318c22d0 , 100 ) = 0x5652318c2320
[0004]           realloc( 0x5652318c2320 , 1000 ) = 0x5652318c23c0
[0005]           realloc( 0x5652318c23c0 , 10000 ) = 0x5652318c27e0
[0006]           realloc( 0x5652318c27e0 , 100000 ) = 0x5652318c4f30
[0007]           free( 0x5652318c4f30 )
[0008]           free( 0x5652318c4f30 )
[0009]     *** DOUBLE_FREE  *** (ignoring)
[0010]           malloc( 100000 ) = 0x5652318c4f30
[0011]           free( 0x5652318c4f30 )
[0012]           free( 0x5652318c4f30 )
[0013]     *** DOUBLE_FREE  *** (ignoring)
[0014]           free( 0x5652318c4f34 )
[0015]     *** ILLEGAL_FREE *** (ignoring)
[0016]
```

```
[0017] Statistics
[0018]   allocated_total      211110
[0019]   allocated_avg        35185
[0020]   freed_total          211110
[0021]
[0022] Memory tracer stopped.
```

위에서 예상한 바와 같이, OS는 100,000 바이트의 메모리 영역에 대하여 주소 `0x5652318c4f30`를 동일하게 배정했음을 확인할 수 있다. 두 번의 double free와 마지막 한 번의 illegal free를 잘 판정했음을 확인할 수 있다.

비슷하게 추가적인 테스트 코드 `test8.c`를 작성하였다.

```c
#include <stdlib.h>

int main(void)
{
  void *a;

  a = malloc(10);
  a = realloc(a, 100);
  a = realloc(a, 1000);
  a = realloc(a, 10000);
  a = realloc(a, 100000);
  free(a);
  free(a);

  a = malloc(100000);
  free(a+4);

  return 0;
}
```

```
stu70@sp01:~/Labs/sp-linklab/linklab/part3$ make run test8
cc -I. -I ../utils -o libmemtrace.so -shared -fPIC memtrace.c ../utils/memlog.c
../utils/memlist.c -ldl
[0001] Memory tracer started.
[0002]           malloc( 10 ) = 0x560663a032d0
[0003]           realloc( 0x560663a032d0 , 100 ) = 0x560663a03320
[0004]           realloc( 0x560663a03320 , 1000 ) = 0x560663a033c0
[0005]           realloc( 0x560663a033c0 , 10000 ) = 0x560663a037e0
[0006]           realloc( 0x560663a037e0 , 100000 ) = 0x560663a05f30
[0007]           free( 0x560663a05f30 )
[0008]           free( 0x560663a05f30 )
[0009]     *** DOUBLE_FREE  *** (ignoring)
[0010]           malloc( 100000 ) = 0x560663a05f30
[0011]           free( 0x560663a05f34 )
[0012]     *** ILLEGAL_FREE *** (ignoring)
[0013]
```

```
[0014] Statistics
[0015]   allocated_total      211110
[0016]   allocated_avg        35185
[0017]   freed_total          111110
[0018]
[0019] Non-deallocated memory blocks
[0020]   block              size       ref cnt
[0021]   0x560663a05f30     100000     1
[0022]
[0023] Memory tracer stopped.
```

Non-deallocated memory blocks를 잘 탐지하였음을 확인할 수 있다.

어려웠던 점

`init` 함수에서 초기에 `dlerror` 함수를 호출해야 하는지 여부를 결정할 때 꽤 많은 reference와 manual pages를 참조하였다.

```
//
// init - this function is called once when the shared library is loaded
//
__attribute__((constructor))
void init(void)
{
  char *error;

  dlerror();

  LOG_START();

  /// 후략
}
```

또한, 공유 라이브러리 로드를 처음 접해보아서, `fini` 함수에서 `RTLD_NEXT` 핸들을 `dlclose`로 닫아야 하는지 헷갈렸다. `dlopen`으로 로드한 핸들만 `dlclose`로 해소하여야 한다. 결과적으로, `fini` 함수에서는 `dlclose`를 호출하지 않았다.

새롭게 배운 점

`dlsym` 함수를 활용하여 load/run-time interposition의 개념을 과제 구현을 통하여 실행 과정을 살펴보았다.

`-shared`와 `fPIC`의 컴파일 옵션으로 position-independent한 `.so` 파일을 생성하고, 이를 링크 후 실행하는 과정을 살펴보았다.