System Programming Lab #3 Report

2019-18499 김준혁

각 블록 구조 및 전반적인 malloc 방법

Allocated Block

HEADER	PAYLOAD	FOOTER
8 bytes	8 * N bytes	8 bytes

Freed Block

HEADER	PREV	NEXT		FOOTER
8 bytes	8 bytes	8 bytes	8 * N bytes	8 bytes

Explicit free list를 이용, size에 descending하게 정렬하고 best fit을 이용하여 memory allocate가 되도록 구현하였다. 이 free list를 head_ptr, tail_ptr를 이용해 관리한다.

함수별 구현 방법

private static functions

static void remove_block(void *ptr) : Explicit free list에서 ptr에 해당하는 블록을 제거
static void insert_block(void *ptr, void *next_to)
: Explicit free list의 next_to 블록 다음에 ptr에 해당하는 블록을 넣는다.
static void *try_coalesce_next(void *ptr)
: ptr 다음의 adjacent한 블록과 coalesce가 가능할 경우, 실행한다.
static void *try_coalesce_prev(void *ptr)
: ptr 이전의 adjacent한 블록과 coalesce가 가능할 경우, 실행한다.
static void insert_root(void *ptr) : ptr 블록을 free list 맨 처음에 넣는다.
static void sort_next(void *ptr)
: ptr 블록을 size에 descending하게 정렬되도록 다음으로 이동하여 옮긴다.
static void sort_prev(void *ptr)
: ptr 블록을 size에 descending하게 정렬되도록 이전으로 이동하여 옮긴다.

list에 원소를 넣고 빼는 것과 그것을 이용해 정렬하는 것은 모두 doubly linked list의 방식을 따라 작성하였다.

int mm_init(void)

Explicit free list를 구현함에 있어, 가장 앞에 있는 head 역할을 하는 32-byte Freed 블록을 만든다. 이 head는 변하지 않으며, 블록은 사실상 다음 포인터를 지정해주는 것에 대해서만 의미를 가진다.

void *mm_malloc(size_t size)

size에 16-byte의 header, footer 공간을 붙여주며, 블록의 크기를 최소 32-byte가 되도록 크기를 조정한다. size에 descending하게 정렬되어 있는 free list에서 size보다 작은 블록을 찾아서 그 전에 있는 블록을 선택하는 방식으로 가장 크기가 비슷한 블록을 찾는다.

여기서 만약 크기에 맞는 블록을 찾지 못한 경우, mem_sbrk로 공간을 만든다. 이때 size는 최소 mem_pagesize(), 필요한 size가 더 클 경우, 그 size에 맞게 공간을 만든다. 이 공간은 list의 맨 처음에 들어가며, 반드시 이 곳에 데이터를 넣을 준비를 하며, 이후 블록을 찾았을 때와 같은 행동을 한다.

블록을 찾으면, 그 블록과의 크기를 비교해서, 같거나 분할 후 남은 공간이 최소 Free block 크기인 32-byte보다 작을 경우, 한 번에 allocate해주고, 그렇지 않은 경우 분할한다. 분할하는 경우에는 기존 블록을 리스트에서 제거해준 이후, 분할하여 그 자리에 다시 넣고, 정렬해준다. 이후 블록의 payload 부분 포인터를 리턴한다.

void mm_free(void *ptr)

ptr 블록의 header와 footer의 allocation-bit를 0으로 바꾸어주고 list의 맨 처음에 넣는다. 이후 앞뒤로 인접한 블록과 coalesce가 가능한 경우 coalesce한다. 이후 sort_next와 sort_prev 모두를 돌려 정렬해준다. 함수는 둘 다 부르지만, 결국 하나만 돌게 되어있다.

void *mm_realloc(void *ptr, size_t size)

ptr가 NULL 인 경우, mm_malloc(size)를 호출하고, 리턴값을 그대로 리턴한다.

size가 0인 경우, mm_free(ptr)를 호출하고, NULL을 리턴한다.

size가 기존 ptr이 차지하는 크기보다 작은 경우(header와 footer를 합하여), 기존의 allocated block을 분할하여, malloc과 비슷한 방식으로 처리한다. 이때, 분할한 블록을 list 맨 처음에 넣은 이후, 다음에 인접한 블록과 coalesce를 시도한다. 이후 정렬해주고, payload 부분 포인터를 리턴한다.

size가 기존보다 큰 경우, 다음 세 가지로 나눈다: 1. 인접한 블록이 freed block이며, 그 공간까지 이용하면 충분한 경우, 2. 인접한 블록이 freed block이며, 공간이 충분하지 않지만, 그 다음 포인터가 현재 heap 범위를 벗어나는 경우, 3. 그 외의 일반적인 경우.

Case 1: 인접한 Freed block을 리스트에서 제거하고, 분할한다. 분할한 Freed block은 malloc에서와 비슷하게 list 처음에 넣은 후 정렬한다. 분할된 block 중 사용할 block은 기존 block과 합한 후, payload 부분 포인터를 리턴한다.

Case 2: 인접한 Freed block을 리스트에서 제거하고, mem_sbrk로 추가적으로 필요한 메모리만큼 공간을 얻은 후, 기존 block, 인접 block, 추가 공간을 모두 합하여 allocated block으로 만들고, payload 부분 포인터를 리턴한다.

Case 3: malloc하고, 해당 위치로 memcpy를 한 이후 free를 하는, 제공된 코드와 거의 같은 방식으로 실행한다. 그리고 payload 부분 포인터를 리턴한다.

어려웠던 점

생각보다 메모리 allocate를 하는 것이 매우 어려웠다. 특히 mdriver를 통한 performance 확인 과정에 있어, 더욱 효율적인 방법을 찾아서 구현하는 것에 대해 수정하는 양에 비해, 그리고 예상한 성능 상승량에 비해 Performance index 상승량이 적어 더욱 피로했다. 또한 디버깅 과정에서 각 block에 대한 property를 추적하면서 여러 번의 포인터를 이용하여 수정을 거치는 것이 굉장히 까다로웠다. 약간의 실수에 payload가 overlap되거나 segment error를 띄우는 경우가 너무 많았다.

realloc을 작성하는 과정에서, 기존에 있던 allocated block(입력으로 들어오는 size보다 많이 작은) 앞뒤로 Freed block이 있는 경우, 이를 최대한 가장자리로 모는 방법을 생각해내고 구현했었다. 하지만 몇 시간을 쓴 이후, 이를 테스트하였을 때, 옮겨진 block들에 대해 그들의 옮겨진 포인터가 반영이 되지 않아 데이터를 가져올 수 없다는 것을 깨달았을 때, 많은 허무감이 느껴졌다. 이를 추적할 수 있는 식으로 구현할 수 있다면 많은 성능 향상이 일어났을 것으로 예상되었는데, 해당 부분은 수정할 수 없는 구역이기 때문에 결국 시간을 허비한 것 같아 안타깝다.

새롭게 배운 점

메모리 allocation에 대한 관리가 매우 까다롭다는 것을 가장 먼저 느끼게 되었던 것 같다. 한 allocation 방법에 대해 case에 따라 성능 차이가 극명하며, fit을 하는 방법, freed list 관리 방법, 더군다나 약간의 변수값을 조절하였을 때에도 차이가 많은 것을 보았을 때, 평균적으로 좋은 방법은 어느 정도 있어도, 특별한 case를 위주로 사용하는 architecture(?)가 있는 경우 그에 맞는 방식으로 allocation 방법을 정하는 것이 좋다는 판단을 내렸다.

이러한 allocation에 다양한 방법이 존재하며, implicit, explicit 등의 freed list, first-fit, best-fit 등 여러 fitting 방법 및 추가적인 allocation 최적화 방법을 알게 되었으며, 이들을 어떻게 작성해야 하는지, 어떤 경우에 특별히 좋은 performance를 보여주는지를 알 수 있던 랩 과제였다고 생각한다.