시스템프로그래밍 : Project #3: Dynamic Memory Allocator

전공: 컴퓨터공학 학년: 3학년 학번: 20201635 이름: 전찬

0. 목차

1. 구현 목표

이번 프로젝트에서는 메모리를 할당하는 방법 중 하나인 Heap 영역의 Dynamic memory allocator을 직접 구현해야 한다. 더 나아가 실제 C언어에서 사용하는 malloc / free / realloc과 동일한 기능을 할 수 있는 함수들을 구현해야 한다. 마지막으로는 throughput과 memory utilization 사이의 최적화를 수행할 수 있는 방법으로 malloc / free / realloc 함수와 이에 연관된 함수들을 최적화하며 Performance 점수를 높일 수 있는 방법으로 구현을 수행해야 한다.

2. 구현 방법 / 구현 설명 / 최적화 과정

- **구현 방법**

Dynamic memory allocator을 구현하는 방법은 다양하게 존재한다. 대표적인 방법으로는 implicit list, explicit list, segregated list 등이 존재한다. 이 중 다른 방법들에 비해 throughput / memory utilization이 다른 방법보다 비교적으로 좋다고 할 수 있는 segregated list 방식으로 Dynamic memory allocator을 구현했다. 또한 기본적인 형태는 LIFO 형태로, 최대한 throughput을 줄이는 형식으로 구현했다.

또한 seg_list의 각 list를 나누는 방식 또한 중요하다. 이는 각 free list가 2^{n-1}~2^n-1 형태의 size를 저장할 수 있도록 설정했다. 따라서 16~31까지는 첫 번째 list, 32~63까지는 두 번째 list 와 같이 block을 저장할 수 있도록 함수들을 구현했다. 구현의 세부 사항들은 아래와 같다.

- **구현 설명**

Segregated list 형태의 Dynamic memory allocator을 구현하기 위해서, 각 크기 마다 Free list를 저장할 수 있는 형태로 구현해야 한다. 하지만 전역 변수로 array를 사용할 수 없기 때문에, void** seg_list를 정의하며, mem_sbrk 함수로 heap 영역에서 여러 free list의 root를 할당하는 방식으로 이를 해결했다. 또한 전역 변수로 void* heap_listp 또한 정의했는데, 이는 heap 영역에서

prologue block을 담당하는 변수로, 교과서의 explicit list 구현을 참고했다. 또한 여러 macro 또한 설정했는데, 교과서에서 explicit list를 구현하는 데에 필요한 macro들을 참고했으며, segregated list를 구현하기 위한 추가적인 macro 들 또한 설정했다. 이에 대한 세부적인 설명은 mm.c code에 주석으로 추가해 놓았다.

이후 mm_init 함수를 구현했다. 위 함수는 mm_malloc / mm_free / mm_realloc을 실행하기 이전, 기본적으로 initialization을 수행하는 함수이다. segregated list을 구현하기 위해서 위에서 정의한 seg_list의 각 원소에 NULL을 할당했으며, 교과서에 있는 구현을 참고하여 prologue / epilogue을 할당해 주었다. 또한 기본적으로 사용할 block을 할당했는데, page size인 CHUNKSIZE 크기의 block을 free list에 할당해 주었다.

또한 Dynamic memory allocator을 구현하기 위한 여러 함수들을 구현했다. 총 6개의 함수를 추가로 구현했는데, 각각의 이름과 routine은 아래와 같다.

- extend_heap

mem_sbrk()를 실제로 call 하여, heap 영역을 확장하는 함수이다. 확장한 영역에 대해, size를 확장한 영역의 block의 header, footer에 대입하며, epilogue을 다시 heap 영역의 마지막에 설정한다. 또한 확장한 block에 대해서도 coalescing을 수행한다.

- coalesce

어떠한 block에 대해, 실제 주소 상의 왼쪽 / 오른쪽 block이 free 되어 있는지 확인하며, free 된 block과 coalescing을 수행하는 함수이다. 왼쪽 / 오른쪽 free block을 먼저 free list에서 제거한 이후, coalescing을 수행하며, 합친 block을 다시 free block에 삽입하는 형태이다.

- insert_block

free된 block을 seg_list의 적절한 index에 삽입하는 함수이다. LIFO 형식으로 segregated list을 구현하기 때문에, 적절한 위치를 찾는 방법은 아래와 같다.

```
int index = 0;
size >>= 4;

while ((index < SEGLEN - 1) && (size > 1)){
    size >>= 1;
    index++;
}
```

<insert_block의 index를 파악하는 코드>

이를 통해, size가 16~31이면 index = 0, 32~63이면 index = 1, 64~127이면 index = 2 와 같은 형식으로 index를 파악해낼 수 있다. 또한 이후 해당 index linked list의 첫 번째 위치에 해당 block을 삽입하는 역할을 수행한다.

- remove_block

remove_block은 malloc, coalesce 등을 위해 free list에서 해당하는 block을 제거하는 함수이다. 이 또한 index를 insert_block과 동일한 코드로 구할 수 있으며, 삭제를 수행한다.

- find_fit

위 함수는 size에 적합한 block을 찾아 return 해주는 함수이다. malloc에서 size에 적합한 block을 찾아내기 위해 사용하는 함수이다. 기본적으로는 LIFO 형식으로, 맨 앞에 있는 block을 찾아 return 해주지만, 만약 해당하는 size가 index = SEGLEN – 1인 경우, 해당 index의 list를 linear search 해야 한다. 왜냐하면 마지막 linked list에는 size의 upper bound가 제한이 없기 때문에, 맨 처음 block이 size가 작더라도, 뒤쪽의 block 중 size가 큰 block이 존재할 수 있기 때문이다. 또한 index를 찾는 방법 또한 위에서 설명한 코드와 약간의 차이가 있으며, 이는 아래와 같다.

```
int index = 0;
size_t searchsize = asize;
searchsize = searchsize - 1;
searchsize >>= 3;

while ((index < SEGLEN - 1) && (searchsize > 1)) {
    searchsize >>= 1;
    index++;
}
```

<find_fit에서 index를 파악하는 코드>

약간의 차이가 존재하는 이유는 LIFO 형태를 위해, 검색 size <= 해당 index의 size 최솟값 이어야 하며, 이를 위해서는 16까지 index = 0, 17~32까지 index = 1, 와 같은 형식으로 검색이 수행되어야 되기 때문이다. 따라서 코드에서 약간의 차이점이 존재한다.

또한 위에서 설명한 것처럼, SEGLEN – 1 에 해당하는 index의 경우, linear search를 수행해야 한다. 이는 간단하게 while loop와 함께 searchlist = NEXT_FREE(searchlist) linked list의 다음 block을 찾아가며, 해당 block의 size >= 필요한 size 인 경우에 해당 block을 사용하면 된다. 만약 size에 적합한 block을 찾지 못했다면, return NULL을 수행한다.

- place

위 함수는 해당하는 block을 malloc할 때, 만약 남는 size가 16 이상이면, block을 분할해주는 함수이다. 분할의 기준이 16인 이유는, free list에서 최소 size가 16 byte이기 때문이다.

위와 같은 기본적인 함수들을 이용해서 mm_malloc / mm_free / mm_realloc을 구현해낼 수 있다. 각각의 routine은 아래와 같다.

- mm_malloc

size에 맞는 block을 free list들에서 find_fit 함수를 통해 찾으며, 만약 찾지 못한 경우 새롭게 heap을 확장하며 block을 할당한다. 할당한 block에 대해 place 함수로 분할을 수행한다.

- mm_free

해당 block의 header, footer에 size와 free(LSB = 0)을 할당하고, coalesce 함수를 수행한다.

- mm_realloc

mm_realloc은 ptr == NULL인 경우에는 mm_malloc을 수행, size == 0 인 경우에는 mm_free을 수행한다. 더 나아가 현재 block의 크기가 realloc size보다 같거나 크다면 block을 그대로 사용하며, 만약 size가 부족한 경우에는 새롭게 mm_malloc(size)로 block을 할당하며 memcopy function으로 데이터를 새로운 block에 복사하는 형식으로 구현했다.

- **최적화 과정**

따라서 위에서 구현한 함수들로 실제 dynamic memory allocator을 수행할 수 있는 mm_malloc, mm_free, mm_realloc function을 구현할 수 있었다. 여기에 더 나아가 위 함수를 최적화할 수 있는 여러가지 방법이 존재하는데, 이는 아래와 같다.

- 코드의 최적화, 즉 function call 등을 최대한 줄이는 형식으로 구현한다.
- seg_list의 최적화, 즉 seg_list의 size division을 여러 형식으로 구현한다.
- SEGLEN의 최적화, 총 저장할 수 있는 linked list를 최적화한다.
- CHUNKSIZE 최적화, heap에서 한 번에 가져오는 block size를 최적화한다.

여기에서 첫 번째 최적화는 구현한 코드의 readability가 크게 저하될 수 있으며, 두 번째 최적화는 만약 seg_list division을 바꾸기 위해서 모든 코드를 바꾸어야 한다는 문제점이 존재한다. 따라서 SEGLEN의 최적화, CHUNKSIZE의 최적화를 통해 가장 높은 performance를 낼 수 있도록 최적화를 수행했다.

우선 첫 번째로, SEGLEN의 최적화를 수행했다. SEGLEN(linked list의 개수)을 10~15로 바꾸어가며 측정해 보았으며, 결과는 아래와 같다.

```
10 : Perf index = 41 (util) + 40 (thru) = 81/100
11 : Perf index = 40 (util) + 40 (thru) = 80/100
12 : Perf index = 38 (util) + 40 (thru) = 78/100
13 : Perf index = 38 (util) + 40 (thru) = 78/100
14 : Perf index = 38 (util) + 40 (thru) = 78/100
15 : Perf index = 38 (util) + 40 (thru) = 78/100
```

<SEGLEN에 따른 performance 측정>

따라서 SEGLEN의 작아질수록 performance가 좋아지는 형태를 파악할 수 있었으며, 따라서 더 작은 size에 대해서도 측정을 수행해 보았다. 결과로 SEGLEN = 7, 8 일 때 83으로 가장 높은 성능을 낼 수 있으며, 이후에는 다시 감소하는 형태임을 파악할 수 있었다. 따라서 SEGLEN = 8로 설정해 주었다.

이후에는 CHUNKSIZE에 대한 최적화를 수행해 주었다. 4096의 초기값에 대해, 여러 배수에 해당하는 size로, 2048, 4096, 8192, 12288 등 여러 숫자를 대입해 보았다. 이 경우들에 대해 CHUNKSIZE = 4096, 8192, 12288 일 때 83으로 최대값이 나왔으며, 이 값들보다 작거나 큰 경우 점점 performance 성능이 떨어졌다. 이후에는 8000, 9000 등 2의 제곱수가 아닌 값들 또한 할당해 보았는데, 9000의 경우에 performance = 84 로 가장 높은 값을 얻어낼 수 있었다. 따라서 CHUNKSIZE는 9000으로 설정해 주었다. 따라서 위 두 가지 방법으로 performance를 78에서 84까지 6% 정도 증가시킬 수 있었다.

3. 개선 사항

위와 같이 여러 함수들을 통해 segregated list 형태의 dynamic memory allocator을 구현하며, 최적화 과정을 통해 performance를 최적화할 수 있었다. 더 나아가, 이번 직접 해보지는 못했지만, 구현한 프로젝트를 더 최적화할 수 있는 여러 개선 사항들이 존재하는데 이는 아래와 같다.

- 위에서 설명한 코드의 최적화, seg_list division의 최적화 수행하기
- explicit list, implicit list를 직접 구현해서 성능 비교해보기
- 각 linked list를 LIFO 형식이 아닌, address ordered, 혹은 size ordered 구현해 비교해보기

등이 존재한다. 하지만 위 개선 사항을 확인하기 위해서는, 새롭게 코드를 구현하거나, 대부분의 함수 구현을 변경해야 한다는 문제점이 존재해서 직접 수행해보지는 못했다. 하지만 위 개선 사항들을 직접 구현해보며, 가장 성능이 좋은 구현을 선택하는 것이 가장 좋은 performance를 얻어낼 수 있을 것 같다.