시스템프로그래밍 실습

과제 3 malloclab

2015-16828 김민규

1. 서론

malloc과 free, realloc을 효율적으로 구현하는 과제였습니다. 저는 seggregated list를 활용하여 free된 블록들을 관리하게끔 구현하였습니다.
전체 heap의 구조는 아래와 같습니다.

0	seggregated list roots	prologue header	block 1	...	block n	epilogue header

처음에는 seggregated list 의 root들이 들어 있습니다. 그 다음 prologue header가 시작을 표시하며, block들이 수록되어 있고 epilogue header로 끝을 표시합니다.
각 block의 구조는 아래와 같습니다.

<allocated blocks>

	WORD
0	header(size = WSIZE*2N, allocation bit = 1)
1	payload start
...	
2N-2	payload end
2N-1	footer(size = WSIZE*2N, allocation bit = 1)

<nonallocated blocks>

	WORD
0	header(size = WSIZE*2N, allocation bit = 0)
1	next free block pointer
2	previous free block pointer
...	
2N-1	footer(size = WSIZE*2N, allocation bit = 0)

블록들은 모두 header와 footer를 가지고 있고, 사이즈와 allocated 여부를 기록합니다. allocated 블록은 안에 payload를 기록해 놓지만, nonallocated block은 1번 word와 2번 word에 각각 다음 nonallocated block의 주소와 이전 nonallocated block의 주소를 기록합니다. 단, 이 때 이전과 다음은 각각의 seggregated list에서의 이전과 다음 nonallocated block입니다.

다시 말해 아래와 같습니다.
```
seg list class 0: root0 -> a1 -> b1 -> c1...
seg list class 1: root1 -> a2 -> b2 -> c2 ...
...
seg list class k-1 : root(k-1) -> a(k-1) ->...
```
위와 같이 사이즈에 따라 seggregated list를 만들었을 때, a1의 next free block pointer에는 b1의 주소가 담기고, previous free block pointer에는 root0가 담깁니다.

2. 본론

2.1. helper function

이 과제를 수행하기 위해 만든 helper function들을 간략히 설명합니다.

2.1.1. get_seg_level(size)

임의의 블록을 seggregated list에 집어넣을 때, 블록의 사이즈에 따라 seggregated list의 어디에 담길 지 class를 결정하는 함수입니다. 이 함수는 size를 받고, 8바이트단위로 쪼갰을 때 몇 개가 나오는지 셉니다. 그 값이 [2^i + 1, 2^(i+1)]에 속할 때 이 블록의 class를 i로 정하고, i를 return합니다. 다만 i가 10이상일 경우는 무조건 10을 return합니다.

2.1.2. get_freeblk_root(size)

임의의 블록을 seggregated list에 집어넣을 때, 그 블록 class의 root를 찾기 위한 함수입니다. size를 받아 get_seg_level로 class를 정하고, 그 class에 맞는 root를 return합니다. root는 heap의 앞쪽에 나란히 위치해 있으므로 heap의 처음을 가리키는 heap_listp에 적당한 값을 더해 root의 위치를 구할 수 있습니다.

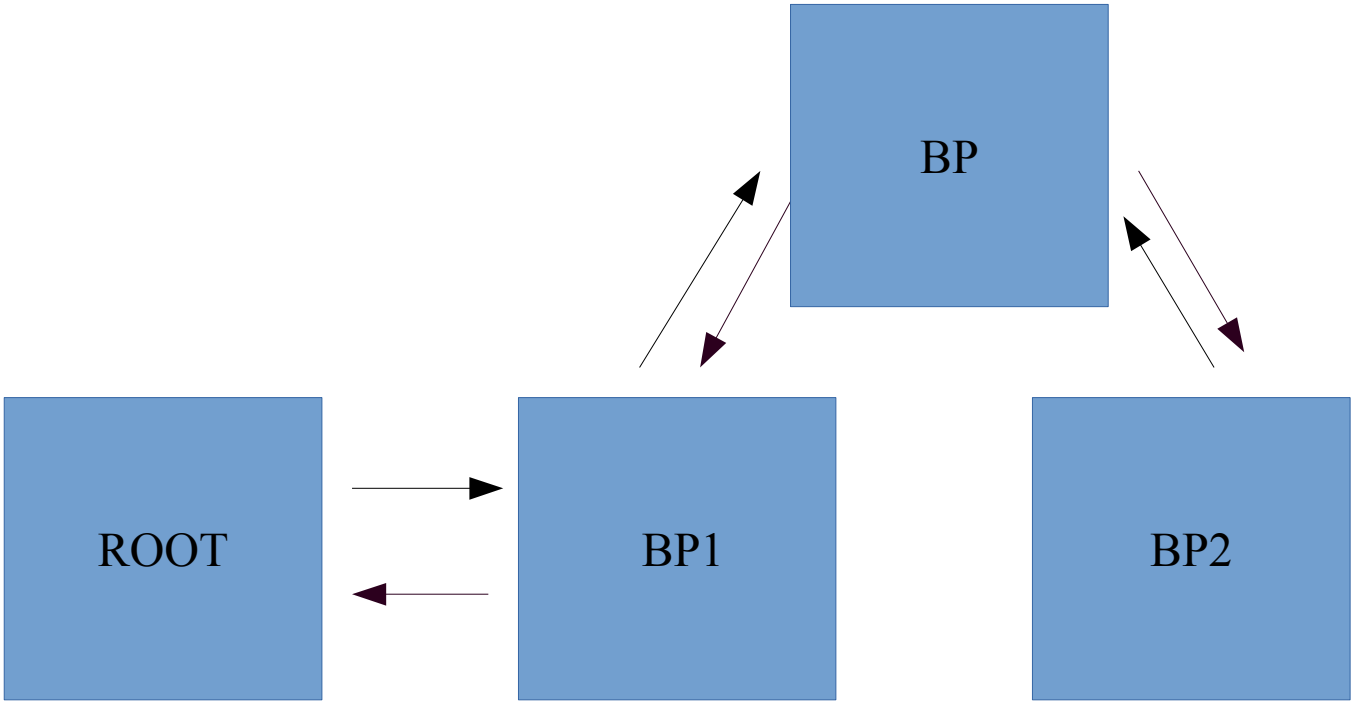
2.1.3. insert_free_block(bp,size)

임의의 블록을 seggregated list에 오름차순으로 집어넣기 위한 함수입니다. 먼저 get_freeblk_root로 root를 찾습니다. 그 다음, 각 블록의 사이즈를 비교해 가며 자신보다 더 큰 블록이 나올 때까지 찾은 후 그 직전에 집어넣습니다. 그런 블록이 없다면 맨 끝에 집어넣습니다.

1) 기존의 seggregated list

root의 next free block pointer는 bp1이고, bp1의 previous free block pointer는 root가 되어 있는 상태입니다. bp2도 마찬가지입니다.

2) BP를 집어 넣었을 때
bp의 사이즈가 bp1의 사이즈보다는 크고 bp2의 사이즈보다는 작은 경우, 아래 그림과 같이 변합니다.



bp1의 next free block pointer를 bp로 바꾸고, bp2의 previous free block pointer도 bp로 바꿉니다. 그 다음 bp의 nenx free block pointer는 bp2으로, previous free block pointer는 bp1로 바꿔 줍니다.

2.1.4. delete_free_block(bp,size)

insert_free_block의 반대입니다. 위의 그림에서 bp를 삭제한다고 하면 다시 2)->1) 의 과정을 수행합니다.

2.1.5. find_fit(size)

size가 주어졌을 때 malloc을 하기 위해 seggregated list에서 크기에 맞는 free block을 찾는 함수입니다.
get_freeblk_root함수로 근을 찾고, 그 근에서 next free block으로 계속해서 찾아나가다가 size보다 큰 free block을 찾으면 그 주소를 return합니다. 찾지 못한다면 다음 class의 근에서 시작해서 반복합니다. 끝까지 진행했지만 찾지 못한다면 NULL을 return합니다.

2.1.6. coalesce(bp)

bp의 coalescing을 진행하는 함수입니다.
Case 1. 앞 뒤 block이 모두 allocated
- 아무 것도 하지 않고 bp를 return합니다.

Case 2. 뒷 block만 free인 경우
- bp와 bp뒷 블록을 모두 delete_free_block으로 seggregated list에서 제거한 다음, 하나의 block으로 만들어 header와 pointer를 새로 기록하고 이 block을 insert_free_block으로 seggregated list에 넣습니다.

Case 3. 앞 block만 free인 경우
- Case 2 와 비슷하게 진행합니다.

Case 4. 앞, 뒤 block 모두 free인 경우
- Case 2 와 비슷하게 진행합니다.

2.1.7. exten_heap(words)

heap을 확장시키는 함수입니다. memlib.h의 mem_sbrk를 통해 새로운 메모리 공간(4096바이트)을 받아 온 후 그것을 free된 block으로 취급해 적절한 header와 footer를 표기합니다. 또한 heap의 마지막에 epilogue header를 넣어 줍니다. free block을 insert_free_block으로 seggregated list에 넣어 준 후 coalesce함수로 coalescing을 진행합니다.

2.1.8. place(bp,asize)

어떤 block의 포인터인 bp를 받아, asize만큼 할당하는 함수입니다. bp가 가리키는 block의 사이즈는 asize보다 큰 것이 보장됩니다.
우선 bp의 allocation bit가 0이라면 bp를 delete_free_block을 통해 seggregated list에서 제거합니다. 그 다음 bp가 가리키는 block의 사이즈에서 asize를 뺀 후의 값이 MIN_BLK_SIZE(16바이트)보다 크다면, 할당하고 남은 메모리를 다시 free해주는 작업이 필요합니다. 만약 그렇지 않고 16바이트보다 작다면, bp의 header와 footer에서 size 값을 asize로 변경해주고 끝납니다.
여기서 MIN_BLK_SIZE가 16바이트인 것은, header와 footer를 쓰려면 8바이트가 있어야 하고, payload또한 1바이트 이상이어야 하므로 alignment를 위해 최소 16바이트 이상 있어야 하기 때문입니다.

2.2. init, malloc, free and realloc

2.2.1 init 함수

N은 seggregated list 의 class 개수입니다.
처음은 (N+3)*16바이트의 공간을 할당받아 초기 작업을 합니다. 이를 그림과 같이 나타내면 아래와 같습니다.

	WORD1(4 bytes)	WORD2(4 bytes)
0	0	
1	NULL	
...	NULL	
N	NULL	
N+1		Prologue header(0X100001)
N+2	Prologue header(0x100001)	Epilogue header(0x1)

빈칸은 할당하지 않은 공간이며, 어느 값이 있든 상관없습니다. Prologue header는 사이즈가 16으로 할당된 블록이 있는 것처럼 행동하여 prologue header보다 더 앞을 조회하지 않도록 합니다. Epilogue header는 heap의 끝을 알려주는 header로, heap size가 늘어난다면 아래와 같이 변합니다.

	WORD1	WORD2
0	0	
1	NULL	
...	NULL	
N	NULL	
N+1		Prologue header(0X100001)
N+2	Prologue header(0x100001)	Header(size, allocation bit)
...	payload	payload
M	Footer(size, allocation bit)	Epilogue header

기존 epilogue header가 있던 자리가 어떤 블록의 header로 바뀌고, payload가 위치한 뒤에 footer가 나오며, epilogue header로 끝을 표시합니다. 이는 extend_heap함수에서 실행되는 사항입니다.

2.2.1 mm_malloc(size)

우선 size가 0이면 비정상적인 request이므로 NULL을 return합니다.
mm_malloc에서는 alignment rule을 지키고 header, footer를 표기하기 위해 size보다 조금 더 큰 block을 할당해야 합니다. 이를 adjusted size, asize라 표기하고 아래와 같은 방식으로 구합니다

```
  if (size <= DSIZE)
    asize = 2*DSIZE;
  else
    asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
```

DSIZE는 8바이트입니다. 앞서 2.1.8에서 언급했던 것처럼 MIN_BLK_SIZE는 16바이트기 때문에 size가 이보다 적다면 asize는 16바이트를 할당해 줍니다.
아니라면, header와 footer를 포함해야 하므로 size에 DSIZE를 더해 주고 그 값에서 alignment rule을 맞춘 값을 asize로 합니다.
그 다음, 만약 find_ft함수로 asize에 맞는 block을 찾는다면 place함수로 할당하고 bp를 return하고 종료합니다. 만약 찾지 못했다면 extend_heap으로 heap사이즈를 늘린 후에 그 포인터에서 place로 할당하고 종료합니다.

2.2.2 mm_free(ptr)

우선 ptr이 NULL이거나 nonallocated block인 경우에는 적절한 에러 메시지를 띄우고 종료합니다.
아니라면, header와 footer를 free block으로 바꿔 준 다음 insert_free_block함수와 coalesce함수를 실행하여 seggregated list에 집어넣습니다.

2.2.3 mm_realloc(ptr,size)

우선 size에 맞는 asize를 구합니다. 그 다음 아래 케이스를 따라 진행합니다.

Case1. 원래의 block 사이즈가 요청된 size보다 큰 경우
- 이 경우는 다른 공간을 할당할 필요가 없습니다. place함수와 비슷하게 진행하면 됩니다. header와 footer의 정보를 바꾸고, 남은 공간이 MIN_BLK_SIZE보다 크다면 free를 진행해주고, 아니라면 종료합니다.

Case2. 원래의 block 사이즈가 요청된 사이즈보다 작을 경우
이 경우는 바로 앞과 바로 뒤의 block을 살펴봅니다.

Case 2.1. next block이 free이고, 원래 block과 다음 block을 합쳤을 때 요청된 사이즈를 할당할 수 있는 경우
- 이 경우는 두 block을 합친 다음 Case1처럼 진행하면 됩니다.

Case 2.2. previous block이 free이고, 원래 block과 previous block을 합쳤을 때 요청된 사이즈를 할당할 수 있는 경우(그리고 Case 2.1.이 아닌 경우)
- 마찬가지로 두 block을 합친 다음, memmove함수를 통해 메모리를 옮겨 줍니다. 그 다음은 Case1처럼 진행하면 됩니다. memcpy대신 memmove를 쓰는 이유는 memmove는 옮기려는 source와 destination이 겹치는 상황에서도 쓸 수 있기 때문입니다.

Case 2.3. previous block, next block 모두 free이고 previous block, 원래 block, next block을 합쳤을 때 요청된 사이즈를 할당할 수 있는 경우(그리고 Case 2.1, 2.2가 아닌 경우)
- Case 2.2.와 마찬가지로 세 block을 합친 다음 memmove를 써 내용을 옮겨주면 됩니다.

Case 2.4. Case 2.1, 2.2, 2.3 모두 아닌 경우
- 이 경우는 mm_malloc으로 새로운 block을 할당한 후에 memcpy를 통해 내용을 복사해줍니다. 그 다음 mm_free로 기존의 block을 free해 줍니다.

2.3. debugging helper 함수

이 함수들은 최종 코드에서 쓰이지는 않지만, 디버깅에 도움을 주는 함수들입니다.

2.2.1 printblocks

heap의 첫 블록부터 시작해 모든 블록을 방문하며 주소, allocate 여부, 사이즈를 출력합니다.

2.2.2 printseglist

seggregated list를 각 class별로 순서대로 방문하며 출력합니다.

2.2.3 mm_check

mm_check는 우선 printblocks에서 한 것과 같이 모든 block을 방문하며 다음과 같은 사항들을 체크합니다.
1. free인 block이 seglist에 속해 있는지
2. 연속인 free block이 있는지
3. 블록이 겹치거나 떨어져 있어, 중간에 block pinter가 NULL 이 되는일이 있는지

또한, seggregated list도 차례로 방문하여 다음과 같은 사항들을 체크합니다.
4. seggregated list에 free가 아닌 block이 있는지

3. 결론과 의견

이 코드를 가지고 테스트 케이스를 실행해 본 결과 86점이 나왔습니다. throughput부분은 40점으로 최고 점수인데, memory utilization에서 40점 중반이 나왔습니다. 특히 realloc부분에서 memory utilization이 안 좋았는데, 아직 이유는 잘 모르겠습니다.