CSE4100:시스템 프로그래밍
Project #3:
Dynamic Memory Allocator
보고서

20190340 강수경

<목차>

1. 개요

(1) 과제물 개요

이번 project의 목표는 c프로그래밍으로 dynamic storage allocator을 구현하는 것이다. malloc, free, realloc 기능을 구현하여 좋은 성능을 가지는 allocator을 만든다.

(2) 요구사항

mm.c 파일의

int mm init(void);

void mm malloc(size t size);

void mm free(void *ptr);

void mm realloc(void *ptr, size t size);

네가지 함수의 기능 구현

heap checker 로 구현한 프로그램의 성능을 checking

(3) 각 함수의 기능

•mm init: Before calling mm malloc mm realloc or mm free, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls mm init to perform any necessary initialization, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise.

• mm malloc: The mm malloc routine returns a pointer to an allocated block payload of at least size bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk.

We will be comparing your implementation to the version of malloc supplied in the standard C library (libc). Since the libc malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.

• mm free: The mm free routine frees the block pointed to by ptr. It returns nothing. This routine is only guaranteed to work when the passed pointer (ptr) was returned by an earlier call to mm malloc or mm realloc and has not yet been freed.

• mm realloc: The mm realloc routine returns a pointer to an allocated region of at least size bytes with the following constraints.

1. if ptr is NULL, the call is equivalent to mm malloc(size);
2. if size is equal to zero, the call is equivalent to mm free(ptr).
3. if ptr is not NULL, it must have been returned by an earlier call to mm malloc or mm realloc. The call to mm realloc changes the size of the memory block pointed to by ptr (the old block) to size bytes and returns the address of the new

block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the realloc request.
The contents of the new block are the same as those of the old ptr block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.

• check heap: heap을 검사하고 일관성이 있는지 검사한다.

2. 개발 내용
implicit list 방식을 이용하여 구현하였다.
implicit list는 묵시적 리스트라는 뜻이고, 가용 블록과 할당된 블록이 서로 섞인 채로 일렬로 배열되어 있는데, 현재 블록에서 이전 블록과 다음 블록이 할당된 블록인지 가용 블록인지는 알 수 없다는 의미다.
가용 블록을 찾는 알고리즘에도 여러 가지가 있는데, malloc lab에서는 2가지를 구현했다.

1)first fit
할당 요청에 맞는 가용 블록을 찾기 위해 블록 리스트의 맨 앞에서 맨 뒤까지 하나씩 검사하면서 할당 요청 크기보다 큰 블록을 찾으면 그 곳에 메모리를 할당하는 방식.
first fit 방식으로 메모리를 할당하면 리스트 앞에 작은 가용 블록들을 남겨두는 경향이 있다.

2) next fit
이전에 메모리를 할당했던 마지막 위치에서부터 요청에 맞는 할당 블록을 서치하는 방식.
리스트의 앞부분이 많은 작은 크기의 조각들로 구성된 경우 매우 빠른 처리속도를 기대할 수 있다.

(1) 구현 함수 개발 방법
<define 내용>
계속 반복되는 내용을 define으로 정의하여 만들었다.

```c
/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
// 선형에서 가장 가까운 배수로 올림?
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

/* Basic constants and macros */
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1<<12)   /* Extend heap by amount (bytes) */

#define MAX(x, y) ((x) > (y)? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))    // 크기와 할당 비트를 통합해서 헤더와 풋터에 저장할 수 있는 값 리턴

/* Read and write a word at address p */
#define GET(p)      (*(unsigned int *)(p))       // p가 참조하는 헤더 리턴
#define PUT(p, val) (*(unsigned int *)(p) = (val))  // p가 가리키는 워드에 val 저장

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)     (GET(p) & ~0x7) // 헤더 사이즈, 할당 비트 리턴
#define GET_ALLOC(p)    (GET(p) & 0x1) // 풋터 사이즈, 할당 비트 리턴

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)    ((char *)(bp) - WSIZE)  // 헤더를 가리키는 포인터 리턴
#define FTRP(bp)    ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE) // 풋터를 가리키는 포인터 리턴

/* Given block ptr bp, comupte address of next and previous blocks */
#define NEXT_BLKP(bp)   ((char *)(bp) + GET_SIZE((char *)(bp) - WSIZE)) // 다음 블록 포인터 리턴
#define PREV_BLKP(bp)   ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE)) // 이전 블록 포인터 리턴
```

int mm_init(void)

힙 리스트를 힙에서 필요한 만큼 가져오기.

필요한 크기의 메모리가 힙 영역에 있는지 확인하고 만약 있다면 그 포인터 주소를, 아니면 -1 출력.(할당기를 초기화하고 성공이면 0, 실패면 -1을 리턴한다.)

먼저 초기 힙 영역을 할당.(mem_sbrk) → 그리고 패딩을 하나 삽입 → prologue header 생성 → prologue footer 생성 → epilogue header 생성 → 초기 힙의 포인터를 prologue header 뒤로 옮긴다. (포인터는 항상 어느 블록이든 header 뒤에 위치한다. 헤더를 읽어서 다른 블록의 값을 가지고오거나 이동을 용이하게 하기 위해서다.) → heap을 한번 특정 사이즈만큼 증가

static void* extend_heap(size_t words)

힙을 초기화 할 때나 추가 힙공간을 요청할 때 호출하는 함수

이전블록이 할당되지 않았다면 받아온 힙여역과 합치는 coalesce 함수 호출

먼저 인자를 words 크기로 받아서, 정렬 기준을 유지하기 위해 2워드의 배수로 반올림. (words%2) ? (words+1) * WSIZE : words * WSIZE; → 그리고 메모리로부터 추가적인 힙 공간을 요청(mem_sbrk)

→ 사이즈를 늘렸으니 그 자리에는 가용 블록이 들어가야 함. 그래야 추후에 malloc을 통해 할당을 요청하면 데이터가 블록에 들어갈 수 있기 때문이다.

→ 새 가용 블록의 header와 footer를 만듬.

→ 그리고 기존 가용리스트의 epilogue header 위치를 조정.(epilogue header는 항상 가용리스트의 끝 부분에 있어야 하기 때문에 위치를 재조정해야 한다.)

static void* first_fit(size_t asize)
맨앞 공간에서부터 하나씩 여유 공간 search
First fit은 리스트의 첫 부분부터 검색하면서 해당하는 size의 블록을 찾는 방법이다.
반복문을 이용하여 구현한다.

static void* next_fit(size_t asize)
이전 포인터에서 뒤로 하나씩 여유공간 search
이전 검색이 종료된 지점에서 리스트의 끝까지 검색하는 부분과 리스트의 시작에서 이전 검색이 종료된 지점까지 검색하는 부분으로 나누어서 구현한다.
할당되지 않고, 사이즈가 큰 블록을 발견하면 해당하는 블록의 포인터를 반환한다.

void* mm_malloc(size_t size)
블록의 크기는 모두 합쳐 8byte 단위가 되도록 설정 한다.
가용 리스트에서 블록을 할당하는 함수
할당기가 요청한 크기를 조정하면 할당할 가용 블록이 가용 리스트에 있는지 탐색한다 → 맞는 블록을 찾으면 요청한 블록을 배치한다.(place) 필요하면 기존 블록을 분할한다. → 맞는 블록을 찾지 못하면 힙을 늘리고 다시 배치한다.

static void place(void* bp, size_t asize)
데이터를 할당할 가용블록의 bp와 배치할 용량을 할당한다. 넣을 위치의 블록 사이즈를 계산한다.

void mm_free(void* ptr)
free 함수는 반환할 블록의 위치를 인자로 받는다. 그리고 해당 블록의 가용 여부를 0(가용상태)으로 만들어버린다. 여기서 핵심은 블록을 가용 리스트에서 제거하는게 아니라 가용상태로 만들어버린다는 것이다. 블록이 반환 되었으니 extend_heap 과 유사하게 블록을 연결하는 함수를 실행한다.

static void* coalesce(void* bp)
(1) 이전과 다음 블록이 모두 할당.
(2) 이전 블록은 할당, 다음 블록은 가용
(3) 이전 블록은 가용, 다음 블록은 할당
(4) 이전 블록과 다음 블록 모두 가용
위의 네 가지 경우들을 구현한다.
위 네가지 경우를 case로 나누어 정리하면
case 1: 앞뒤 사이즈 둘다 모두 0이 아닌 경우
case 2: 뒤가 0인 경우
case 3: 앞이 0인 경우
case 4: 앞뒤 사이즈가 모두 0인 경우

void* mm_realloc(void* ptr, size_t size)
크기를 조정할 블록의 위치와 요청 사이즈를 인자로 받는다. 그 후 malloc(size) 을 통해 요청 사

이즈만큼 블록을 할당한다.

3. 개발 결과

1)first fit 구현시

44 (util) + 10 (thru) = 55/100 의 결과가 나왔다.

2)next fit 구현시

42 (util) + 40 (thru) = 82/100 의 결과가 나왔다.

(아래 결과사진 첨부)

```
cse20190340@cspro:~/prj3-malloc$ ./mdriver
[20190340]::NAME: Sookyung Kang, Email Address: crab11@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Perf index = 44 (util) + 10 (thru) = 55/100
cse20190340@cspro:~/prj3-malloc$ make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
mm.c:134:14: warning:  first_fit defined but not used [-Wunused-function]
 static void* first_fit(size_t asize)
              ^
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o fti
mer.o
cse20190340@cspro:~/prj3-malloc$ ./mdriver
[20190340]::NAME: Sookyung Kang, Email Address: crab11@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Perf index = 42 (util) + 40 (thru) = 82/100
```

mdriver를 통해 성능을 비교해보면, first fit 보다는 next fit 방법이 좋다는 것을 알 수 있다.

first fit 방식은 처음부터 블록을 탐색하므로 utilization은 다소 증가하지만 throughput은 좋지 않다. next fit을 이용하면 throughput이 조금 향상되어 전체적인 성능이 좋아진다.