Multicore Programming Project 3

담당 교수 : 최재승

이름 : 강상원

학번 : 20191559

1. 개발 목표

C 프로그램을 위한 dynamic memory allocator를 직접 구현해본다; 효율적이고 정확하며, 빠른 방향으로 malloc, free, realloc을 직접 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

```
int mm init(void);
void mm malloc(size t size);
void mm free(void *ptr);
void mm realloc(void *ptr, size t size);
```

B. 개발 내용

mm.h에 선언되어 있는 위의 함수들을 mm.c에서 구현한다.

- mm_init

  mm_malloc, mm_realloc, mm_free를 호출하기 전에, 초기 힙 영역 할당 등 기초 작업을 담당한다. 수행 중 오류시 -1 반환, 그 외 0 반환.

- mm_malloc

  전달받은 크기로 최소 payload만큼 할당한다. libc malloc과 동일하게, 8-byte 정렬된 포인터를 항상 반환한다.

- mm_free

  전달받은 ptr 포인터에 있는 메모리 블럭을 할당 해제한다.

- mm_realloc

  전달받은 포인터가 NULL이면, mm_malloc과 동일한 작업 수행, size가 0이면 mm_free와 동일. ptr이 가리키는 메모리 블럭을 size 바이트 크기로 변경하고 다시 주소를 반환한다.

C. 개발 방법

✓ subroutines, structs, global variable 설명

- `static void *heap_listp = NULL; // Empty list`

빈 블럭의 연결 리스트 값 포인터를 제공한다.

- `static char *init_heap_space(void);`

```
/**
 * @brief Initialize list pointer, heap space
 * @return space pointer or -1 on error
 */
```

mem_sbrk를 호출하여 4*WSIZE만큼 힙 영역을 초기화한다. 만약 할당에 실패하면, -1을 반환하고 이외에는 할당된 포인터를 반환한다.

- `static void create_heap(char *heap_s);`

```
/**
 * @brief Create heap structure with prologue and epilogue
 * @param heap_s heap space pointer
 */
```

힙 영역 포인터를 인자로 받아서 프롤로그, 에필로그 형식을 포함한 힙 영역을 생성한다.

- `static void *extend_heap(size_t words);`

```
/**
 * @brief Extend the heap with a new free block
 * @param words size of the new free block
 * @return pointer of the new free block
 */
```

힙 영역을 새로운 빈 블럭을 추가해 확장한다. 새로 할당할 블럭 사이즈를 인자로 받아, 할당된 블럭의 주소를 반환한다.

- `static char *extend_heap_if_needed(char *bp, size_t asize, int *left);`

```
/**
 * @brief Extend heap if needed, for a given block pointer and size
 * @param bp Block pointer
 * @param asize size of block
 * @param left size left in current block, if applicable
 * @return Pointer to block / NULL
 */
```

주어진 블럭 포인터와 크기를 받아, 필요한 경우 힙 영역을 확장한다. 현재 블럭에 남은 영역의 크기, 블럭의 크기 등을 종합적으로 판단한다.

- `static char *fit_block(size_t asize);`

```
/**
 * @brief Get an adequate block for the size
 * @param size Size of block
```

```
 * @return Pointer to adequate block or NULL
 */
```

사이즈에 맞는 크기의 블럭을 찾아 제공한다. 기본적으로 8바이트 정렬을 따르며, 오름차순으로 정렬되어 있는 빈 영역들을 순차적으로 탐색하면서 알맞은 빈 영역을 찾는다. 찾은 빈 영역을 반환한다.

- `static void *place(void *bp, size_t asize);`

```
/**
 * @brief Place block of asize bytes at start of free block bp
 * and split if remainder would be at least minimum block size
 * @param bp pointer of block
 * @param asize size of block
 * @return pointer of placed block
 */
```

인자로 받은 `asize` 바이트만큼을 빈 블럭 `bp` 앞에 두고, 만약 나머지가 최소 요구 블럭 사이즈보다 작으면 `split` 불가로 새로운 블럭 할당, 이외에는 `split` 작업을 한다.

- `static void insertion(size_t size, char *ptr);`

```
/**
 *  @brief Find free blocks, and insert (Explicit list
implementation)
 * @param size size of block
 * @param ptr pointer of block
 */
```

Explicit list으로 구현된 영역을 탐색하며, 인자로 받은 size를 할당할 수 있는 블럭을 발견할 때까지 순차적으로 탐색한다. 기존 빈 블럭은 increasing order로 할당되어 있어 탐색의 효율성을 추구하였다. 크게 리스트의 처음이 아닌 경우, 처음인 경우로 나누어 맨 앞인 경우 initial 블럭 값을 바꾸고, 아닌 경우 삽입 시 앞, 뒤 연결 리스트를 수정한다.

- `static void deletion(char *ptr);`

```
/**
 *  @brief Find blocks by case, and delete (Explicit list
implementation)
 * @param ptr pointer to delete
 */
```

인자로 받은 ptr이 가리키는 블럭을 할당 해제한다. 크게 리스트의 처음이 아닌 경우, 처음인 경우로 나누어 맨 앞인 경우 initial 블럭 값을 바꾸고, 아닌 경우 삽입 시 앞, 뒤 연결 리스트를 수정한다.

- `int mm_init(void);`

앞서 설명한 init_heap_space()을 통해 초기 힙 영역을 설정한다. 이후 `create_heap()`을 이용해 프롤로그 헤더, 푸터, 에필로그 헤더 등을 설정한다. 모듈화를 통해 프로그램 구조에 변경이 있을 시 dependency를 최소화하고자 하였다.

- `void *mm_malloc(size_t size);`

```
/**
 * @brief Allocate block of size bytes
 * @param size Size of block
 * @return Pointer to block or NULL
 */
```

인자로 받은 size 바이트만큼 힙 메모리 영역을 찾아 할당한다. 전달받은 크기로 최소 payload만큼 할당한다. `libc malloc`과 동일하게, 8-byte 정렬된 포인터를 항상 반환하도록 작성하였다. 앞서 설명한 `fit_block()`으로 알맞은 블럭을 찾고, `extend_heap_if_needed()`으로 필요한 경우 힙 영역을 확장한다. 할당에 실패한 경우, `NULL` 값을 반환하며 그 외의 경우 후술할 `place()` 함수의 반환값인 할당 위치를 반환한다. (빈 블럭 리스트의 첫 부분에 삽입)

- `void mm_free(void *bp);`

```
/**
 * @brief Allocate block of size bytes
 * @param size Size of block
 * @return Pointer to block or NULL
 */
```

전달받은 ptr 포인터에 있는 메모리 블럭을 할당 해제한다. `BLOCK_SIZE()` 매크로 함수로 포인터가 가리키는 블럭의 크기를 측정한 뒤에, 헤더와 푸터를 빈 블럭에 맞게 설정한다.

- `void *mm_realloc(void *ptr, size_t size);`

```
/**
 * @brief Reallocate memory block pointed by ptr
 * @param ptr Pointer of memory block
 * @param size Size of memory block
 * @return Reallocated block pointer or NULL
 */
```

전달받은 포인터가 NULL이면, `mm_malloc`과 동일한 작업 수행, size가 0 아면 `mm_free`와 동일. ptr이 가리키는 메모리 블럭을 `size` 바이트 크기로 변경하고 다시 주소를 반환한다.

ALIGN()을 통해 8-byte 정렬을 한 `size` 크기만큼 공간을 설정한다. 현재 있는 공간의 크기와, 변경하고자 하는 크기를 비교해 변경 후 크기가 더 큰 경우에만 확장 작업을 수행한다. 앞서 구현한 `mm_malloc()`을 통해 새 메모리를 할당하고 기존 데이터를 이곳에 옮긴 다음 기존 영역을 `mm_free()`로 할당 해제한다. 만약 다음 블럭이 빈 공간일 경우, `extend_heap_if_needed()`로 영역을 확장하는 방식으로 reallocation을 수행한다.

- `static void *place(void *bp, size_t asize);`

```
/**
 * @brief Place block of asize bytes at start of free block bp
 * and split if remainder would be at least minimum block size
 * @param bp pointer of block
 * @param asize size of block
 * @return pointer of placed block
 */
```

인자로 받은 `asize` 바이트만큼 빈 블럭 연결 리스트의 처음에 연결시킨다. 크게 세 가지 경우로 나눌 수 있는데, 먼저 initial 블럭 값이 place하고자 하는 용량을 할당할 수 있을만큼 공간이 있다면, split하고 이곳에 집어넣는다. 그렇지 않은 경우, 맨 앞에 블럭을 삽입한다. `PLACE_THRESHOLD = ALIGNMENT << 3` 즉, 현재 정렬 기준값의 3배 이상(8 byte alignment로 설정하였으므로 24 byte)인 경우, 큰 메모리 값을 리스트 맨 앞에 집어 넣는 것은 탐색, 메모리 할당 관점에서 비효율적이므로 맨 뒤로 뺀다. (덩치 큰 부분들을 최대한 배제)

- `static void *coalesce(void *bp);`

```
/**
 * @brief Coalesce free blocks
 * @param bp pointer of block
 * @return pointer of coalesced block
 */
```

빈 블럭들을 찾아 coalesce(병합)한다. 앞 뒤 모두 비어 있지 않은 경우, 뒤가 비어 있는 경우, 앞만 비어 있는 경우, 둘 다 비어 있는 경우 이렇게 4가지 경우로 나눠 병합 작업을 진행한다. 앞서 설명한 `extend_heap`에서 이를 호출하여 빈 공간을 최대한 병합하는 작업을 선행한다.

다음은 서술한 함수들의 코드 가독성, redundancy 및 속도를 개선하기 위해 쓰인 매크로 함수들이다.

- `#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)`

8바이트 정렬을 위한 매크로 함수이다. `0x7`은 `111…111000`으로 나타내

어지며, 이의 NOT 값이므로 000…00111이다. 전달받은 size에 AND 연산을 취하므로 맨 뒤 세 비트가 000이 되어 8바이트 정렬을 수행할 수 있다.

- `#define NEXT_BLK(ptr) (*(char **)(ptr))`

  연결 리스트의 다음 블럭 포인터 값을 반환한다.

- `#define PREV_BLK(ptr) (*(char **)(ptr + WSIZE))`

  연결 리스트의 이전 블럭 포인터 값을 반환한다.

- `#define BLOCK_SIZE(ptr) (GET_SIZE(HDRP(ptr)))`

  구현에 있어 HDRP()의 GET_SIZE()값을 호출하는 구문이 자주 쓰여 간소화를 위해 한번에 블럭 크기를 찾을 수 있게 하였다.

- `#define SET(p, ptr) (*(uintptr_t *)(p) = (uintptr_t)(ptr))`

  ptr 값을 p에 assign한다. uintptr_t 자료형은 포인터의 주소를 저장하는데 사용되고, 안전한 포인터 선언 방법을 제공하여 사용하였다.

✓ 수행 결과 검증

driver 프로그램으로 performance index(P) 점수를 매기는 방식은 다음과 같다.

$$P = wU + (1-w)\min\left(1, \frac{T}{T_{libc}}\right)$$

$$U: space\ utilization$$

$$T: Throughput$$

$$T\_libc: libc의\ malloc\ throughput$$

다음은 mdriver -v 를 실행했을 때의 결과이다.

```
cse20191559@cspro:~/malloc$ ./mdriver -v
[20191559]::NAME: SangWon Kang, Email Address: me@kevink1113.com
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000706  8067
 1       yes   99%    5848  0.000459 12738
 2       yes   99%    6648  0.001419  4684
 3       yes   99%    5380  0.000688  7814
 4       yes   99%   14400  0.000357 40291
 5       yes   95%    4800  0.012066   398
 6       yes   95%    4800  0.011671   411
 7       yes   61%   12000  0.090305   133
 8       yes   88%   24000  0.027701   866
 9       yes   99%   14401  0.000122117944
10       yes   98%   14401  0.000159 90402
Total          94%  112372  0.145655   771

Perf index = 56 (util) + 40 (thru) = 96/100
cse20191559@cspro:~/malloc$
```

✓ Flow Chart로 표현한 allocator 디자인 구조.