System Programming Project 4

담당 교수 : 박성용

이름 : 박민준

학번 : 20212020

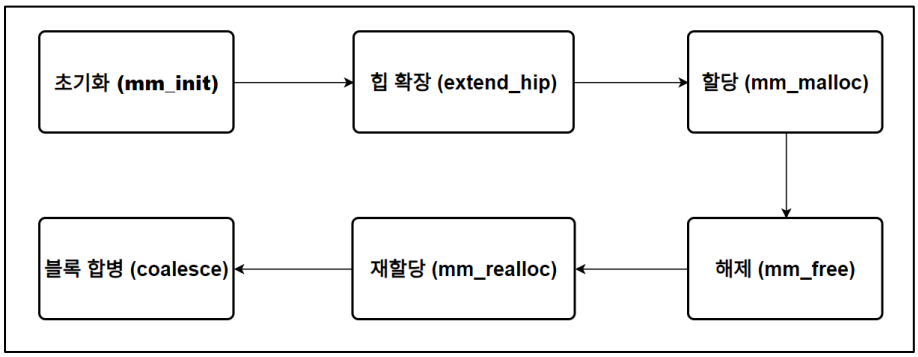
1. Design of my allocator

1.1. 할당기의 설계 개요

- 이번 프로젝트의 동적 메모리 할당기는 분리 적재 목록(segregated free list)을 사용하여 메모리 블록을 관리한다. 이는 서로 다른 크기의 블록을 별도의 목록으로 관리하여 할당과 해제를 보다 효율적으로 수행하게 한다. 또한, 메모리 조각화를 최소화하고, 효율적인 메모리 재사용을 목표로 한다.

1.2. 설계 흐름도

- 다음은 메모리 할당기 설계의 간단한 기본 흐름도이다.



2. Description of subroutines

2.1. 초기화 함수 (mm_init)

- mm_init 함수는 할당기의 초기화를 담당한다. 힙을 초기화하고, 초기 빈 힙을 설정한 뒤, extend_heap 함수를 호출하여 힙을 초기 크기로 확장한다. 분리 적재 목록 배열도 초기화한다.

```c
int mm_init(void)
{
    void** seg_lists[TOTAL_LIST] = {
        &seg_list_1, &seg_list_2, &seg_list_3, &seg_list_4,
        &seg_list_5, &seg_list_6, &seg_list_7, &seg_list_8,
        &seg_list_9, &seg_list_10, &seg_list_11, &seg_list_12,
        &seg_list_13, &seg_list_14, &seg_list_15, &seg_list_16,
    };

    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));
    heap_listp += (2 * WSIZE);

    int i = 0;
    while (i < TOTAL_LIST) {
        *seg_lists[i] = NULL;
        i++;
    }

    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    return 0;
}
```

2.2. 힙 확장 함수 (extend_heap)

- extend_heap 함수는 힙을 확장하여 새로운 메모리 블록을 추가한다. 주어진 워드 단위 크기를 바이트 단위로 변환하고, 필요시 2워드 정렬을 맞춘다. 새로 할당된 블록의 헤더와 푸터를 설정하고, 힙의 에필로그 헤더를 업데이트한다. 새로 할당된 블록을 합병(coalesce)하여 반환한다.

```c
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((bp = mem_sbrk(size)) == (void *)-1)
        return NULL;

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    return coalesce(bp);
}
```

2.3. 블록 합병 함수 (coalesce)

- coalesce 함수는 인접한 빈 블록들을 합쳐 하나의 큰 블록으로 만든다. 이전 블록과 다음 블록의 할당 상태를 확인하여 다음과 같은 4가지 경우로 나누어 처리한다.

1) 이전 블록과 다음 블록 모두 할당된 상태.

2) 이전 블록은 할당되었고 다음 블록은 비어 있는 상태.

3) 이전 블록은 비어 있고 다음 블록은 할당된 상태.

4) 이전 블록과 다음 블록 모두 비어 있는 상태.

- 각 경우에 따라 블록을 합병하고, 새로운 블록을 적절한 분리 적재 목록에 삽입한다.

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) {
        insert_node(bp, size);
        return bp;
    }

    if (prev_alloc && !next_alloc) {
        delete_node(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    } else if (!prev_alloc && next_alloc) {
        delete_node(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    } else {
        delete_node(PREV_BLKP(bp));
        delete_node(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    insert_node(bp, size);
    return bp;
}
```

2.4. 메모리 할당 함수 (mm_malloc)

- mm_malloc 함수는 주어진 크기의 메모리를 할당한다. 요청한 크기를 조정하고, 적절한 블록을 찾는다. 적합한 블록을 찾지 못하면 힙을 확장하여 새로운 블록을 할당한다.

```c
void *mm_malloc(size_t size)
{
    size_t asize;
    size_t extendsize;
    char *bp;

    if (size == 0)
        return NULL;

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

    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

2.5. 메모리 해제 함수 (mm_free)

- mm_free 함수는 주어진 포인터가 가리키는 메모리 블록을 해제한다. 블록의 헤더와 푸터를 업데이트하여 블록을 비어 있는 상태로 표시한 후, coalesce 함수를 호출하여 인접한 빈 블록들을 합병한다.

```c
void mm_free(void *bp)
{
    if (bp == NULL)
        return;

    size_t size = GET_SIZE(HDRP(bp));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}
```

2.6. 메모리 재할당 함수 (mm_realloc)

- mm_realloc 함수는 주어진 포인터가 가리키는 메모리 블록의 크기를 재조정한다. 새로운 크기에 따라 블록을 확장하거나, 새로운 블록을 할당하고 기존 데이터를 복사한다.

```
void *mm_realloc(void *ptr, size_t size)
{
    if ((int)size < 0)
        return NULL;
    else if ((int)size == 0) {
        mm_free(ptr);
        return NULL;
    } else if (size > 0 && ptr == NULL) {
        return mm_malloc(size);
    }

    void *new_ptr = ptr;
    size_t new_size = size + REALLOC_BUFFER;
    int remainder;
    size_t extendsize;
    size_t block_size = GET_SIZE(HDRP(ptr));

    if (new_size <= block_size) {
        return ptr;
    } else {
        if (!GET_ALLOC(HDRP(NEXT_BLKP(ptr))) || !GET_SIZE(HDRP(NEXT_BLKP(ptr)))) {
            remainder = block_size + GET_SIZE(HDRP(NEXT_BLKP(ptr))) - new_size;
            if (remainder < 0) {
                extendsize = MAX(-remainder, CHUNKSIZE);
                if (extend_heap(extendsize / WSIZE) == NULL)
                    return NULL;
                remainder += extendsize;
            }
```

```
            delete_node(NEXT_BLKP(ptr));
            PUT(HDRP(ptr), PACK(new_size + remainder, 1));
            PUT(FTRP(ptr), PACK(new_size + remainder, 1));
        } else {
            new_ptr = mm_malloc(new_size);
            memcpy(new_ptr, ptr, MIN(size, new_size));
            mm_free(ptr);
        }
    }
    return new_ptr;
}
```

2.7. 블록 삽입 함수 (insert_node)

- insert_node 함수는 새로 생성된 빈 블록을 적절한 분리 적재 목록에 삽입하는 역할을 한다. 블록의 크기에 따라 적절한 리스트를 선택하고, 선택된 리스트에 새 블록을 추가한다.

1) 리스트 선택: 블록의 크기에 따라 적절한 분리 적재 목록을 선택한다. 크기가 클수록 리스트의 인덱스가 커진다.

2) 블록 검색: 선택된 리스트에서 적절한 위치를 찾기 위해 블록을 검색한다.

3) 블록 삽입: 새 블록을 리스트에 삽입한다. 삽입 위치는 insert_ptr과 search_ptr 포인터에 따라 결정된다.

```c
static void insert_node(void *bp, size_t size)
{
    void** seg_lists[TOTAL_LIST] = {
        &seg_list_1, &seg_list_2, &seg_list_3, &seg_list_4,
        &seg_list_5, &seg_list_6, &seg_list_7, &seg_list_8,
        &seg_list_9, &seg_list_10, &seg_list_11, &seg_list_12,
        &seg_list_13, &seg_list_14, &seg_list_15, &seg_list_16,
    };

    int list = 0;
    void *search_ptr = bp;
    void *insert_ptr = NULL;

    while ((list < TOTAL_LIST - 1) && (size > 1)) {
        size >>= 1;
        list++;
    }

    search_ptr = *seg_lists[list];
    while ((search_ptr != NULL) && (size > GET_SIZE(HDRP(search_ptr)))) {
        insert_ptr = search_ptr;
        search_ptr = SUCC(search_ptr);
    }

    if (search_ptr != NULL) {
        if (insert_ptr != NULL) {
            SUCC(bp) = search_ptr;
            PRED(search_ptr) = bp;
            SUCC(insert_ptr) = bp;
            PRED(bp) = insert_ptr;
        } else {
            SUCC(bp) = search_ptr;
            PRED(search_ptr) = bp;
            PRED(bp) = NULL;
            *seg_lists[list] = bp;
        }
    } else {
        if (insert_ptr != NULL) {
            SUCC(insert_ptr) = bp;
            PRED(bp) = insert_ptr;
            SUCC(bp) = NULL;
        } else {
            SUCC(bp) = NULL;
            PRED(bp) = NULL;
            *seg_lists[list] = bp;
        }
    }
}
```

2.8. 블록 제거 함수 (delete_node)

- delete_node 함수는 분리 적재 목록에서 블록을 제거하는 역할을 한다. 블록의 크기에 따라 적절한 리스트를 선택하고, 선택된 리스트에서 해당 블록을 제거한다.

1) 리스트 선택: 블록의 크기에 따라 적절한 분리 적재 목록을 선택한다. 크기가 클수록 리스트의 인덱스가 커진다.

2) 블록 제거: 선택된 리스트에서 해당 블록을 제거한다. 블록의 이전 포인터(PRED(bp))와 다음 포인터(SUCC(bp))를 사용하여 리스트를 갱신한다.

```
static void delete_node(void *bp)
{
    void** seg_lists[TOTAL_LIST] = {
        &seg_list_1, &seg_list_2, &seg_list_3, &seg_list_4,
        &seg_list_5, &seg_list_6, &seg_list_7, &seg_list_8,
        &seg_list_9, &seg_list_10, &seg_list_11, &seg_list_12,
        &seg_list_13, &seg_list_14, &seg_list_15, &seg_list_16,
    };

    int list = 0;
    size_t size = GET_SIZE(HDRP(bp));

    while ((list < TOTAL_LIST - 1) && (size > 1)) {
        size >>= 1;
        list++;
    }

    if (PRED(bp) != NULL) {
        SUCC(PRED(bp)) = SUCC(bp);
    }
    else {
        *seg_lists[list] = SUCC(bp);
    }
    if (SUCC(bp) != NULL) {
        PRED(SUCC(bp)) = PRED(bp);
    }
}
```

3. Description of structs

3.1. team_t

- 팀 정보를 저장하는 구조체이다.

```
typedef struct {
    char *student_id; /* ID1+ID2 or ID1 */
    char *name1;    /* full name of first member */
    char *id1;      /* login ID of first member */
} team_t;
```

3.2. 글로벌 변수

- heap_listp: 힙의 시작 포인터

- segregated_list_1 ~ 16: 분리 적재 목록 배열

```c
/* Global variables */
static char *heap_listp = 0;
static void* seg_list_1, *seg_list_2, *seg_list_3, *seg_list_4;
static void* seg_list_5, * seg_list_6, * seg_list_7, * seg_list_8;
static void* seg_list_9, * seg_list_10, * seg_list_11, * seg_list_12;
static void* seg_list_13, * seg_list_14, * seg_list_15, * seg_list_16;
```

4. Description of global variables that I knewly define in my code

4.1. 기본 상수 및 매크로

- WSIZE: 워드 크기 (4 바이트)

- DSIZE: 더블 워드 크기 (8 바이트)

- CHUNKSIZE: 힙을 확장하는 기본 크기 (4096 바이트)

- LISTLIMIT: 분리 적재 목록의 개수 (20개)

- ALIGNMENT: 정렬 요구사항 (8 바이트)

- REALLOC_BUFFER: 재할당 시 단편화를 줄이기 위한 버퍼 크기

```c
/* Basic constants and macros */
#define WSIZE       4       /* Word and header/footer size (bytes) */
#define DSIZE       8       /* Double word size (bytes) */
#define CHUNKSIZE   (1<<12) /* Extend heap by this amount (bytes) */
#define TOTAL_LIST   16      /* Number of segregated lists */
#define ALIGNMENT   8       /* Alignment requirement */
#define REALLOC_BUFFER  (1<<7) /* Buffer for realloc to reduce fragmentation */
```

4.2. 매크로 함수

- MAX(x, y): 두 값 중 더 큰 값을 반환한다.

- MIN(x, y): 두 값 중 더 작은 값을 반환한다.

- PACK(size, alloc): 주어진 크기와 할당 비트를 결합하여 헤더/푸터 값을 생성한다.

- GET(p), PUT(p, val): 주어진 주소 p에서 값을 읽고 쓰는 매크로.

- GET_SIZE(p), GET_ALLOC(p): 주어진 주소 p에서 블록의 크기와 할당 비트를 추출하는 매크로.

- HDRP(bp), FTRP(bp): 블록 포인터 bp에서 헤더와 푸터의 주소를 계산하는 매크로.

- NEXT_BLKP(bp), PREV_BLKP(bp): 다음과 이전 블록의 주소를 계산하는 매크로.

```c
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) < (y) ? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc)  ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p)       (*(unsigned int *)(p))
#define PUT(p, val)  (*(unsigned int *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

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

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

5. Conclusion

- 해당 보고서는 동적 메모리 할당기 프로젝트의 주요 설계 및 구현 세부 사항을 다룬다. 본 프로젝트의 목적은 malloc, free, realloc 함수를 포함한 메모리 할당기를 올바르게, 효율적으로, 빠르게 동작하도록 구현하는 것이다. 이를 위해 분리 적재 목록(segregated free list) 방식을 사용하여 메모리 블록을 관리하고, 이를 통해 메모리 조각화 문제를 최소화하고 효율적인 메모리 재사용을 달성하였다.

- 프로젝트 주요 성과는 다음과 같다.

1) 효율적인 메모리 관리: 분리 적재 목록을 사용하여 서로 다른 크기의 블록을 효율적으로 관리함으로써 메모리 조각화를 줄이고, 메모리 할당 및 해제의 성능을 향상시켰다.

2) 안정적인 힙 확장: extend_heap 함수는 필요한 경우 힙을 안정적으로 확장하여 충분한 메모리를 제공한다. 이는 초기화 과정에서와 할당 요청 시 적절히 동작한다.

3) 효과적인 블록 합병: coalesce 함수는 인접한 빈 블록을 합병하여 큰 블록으로 만들어 메모리 단편화를 줄이고, 이후 할당 요청을 효율적으로 처리할 수 있도록 하였다.

4) 최적화된 재할당: mm_realloc 함수는 기존 블록의 크기를 효율적으로 조정하고, 필요시 새로운 블록을 할당하여 데이터를 복사함으로써 메모리 사용을 최적화하였다.

- 해당 프로젝트를 통해 동적 메모리 할당기의 설계와 구현을 통해 메모리 관리의 효율성을 높이고, 성능을 최적화하는 방법을 탐구하였다. 프로젝트에서 사용한 분리 적재 목록 방식은 메모리 조각화를 줄이고, 메모리 할당과 해제를 보다 효율적으로 처리하는 데 기여하였다.

- 프로젝트 코드 구현 후 make 명령어를 통해 실행 파일을 만든 뒤, ./mdriver -V 명령어를 입력하면 그 결과는 다음과 같다.

```
cse20212020@cspro:~/SP/Lab4/prj4-malloc$ make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
cse20212020@cspro:~/SP/Lab4/prj4-malloc$ ./mdriver -V
[20212020]::NAME: Minjun Park, Email Address: mj4863@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.004399  1294
 1       yes   97%    5848  0.003713  1575
 2       yes   96%    6648  0.007845   847
 3       yes   98%    5380  0.004088  1316
 4       yes   66%   14400  0.012002  1200
 5       yes   93%    4800  0.015173   316
 6       yes   90%    4800  0.017307   277
 7       yes   55%   12000  0.258625    46
 8       yes   51%   24000  0.738286    33
 9       yes   99%   14401  0.003376  4265
10       yes   66%   14401  0.003999  3601
Total          83%  112372  1.068814   105

Perf index = 50 (util) + 7 (thru) = 57/100
```

- mdriver 실행파일을 통해 utilization 효율과 throughput 효율을 계산한 최종 효율성 점수를 확인할 수 있다. 이를 통해 동적 할당을 통한 메모리가 얼마나 효율적으로 사용되고 있는지 쉽게 파악할 수 있었다.