2023 시스템 프로그래밍

- Malloc Lab -

제출일자	2023. 11. 28.
분 반	00
이 름	김재덕
학 번	202104340

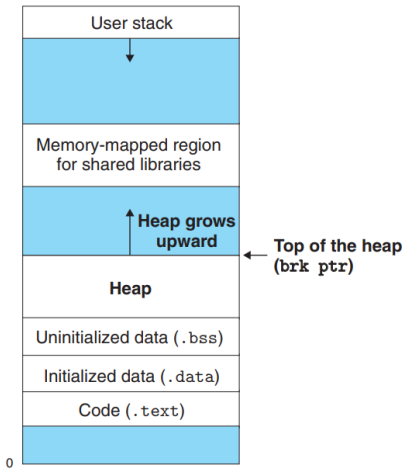
Naive

```
~/malloclab-handout $ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3791.3 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes     94%      10  0.000000 58239 ./traces/malloc.rep
   yes     77%      17  0.000000 44206 ./traces/malloc-free.rep
   yes    100%      15  0.000000 33199 ./traces/corners.rep
 * yes     71%    1494  0.000025 58651 ./traces/perl.rep
 * yes     68%     118  0.000002 54485 ./traces/hostname.rep
 * yes     65%   11913  0.000224 53071 ./traces/xterm.rep
 * yes     23%    5694  0.000107 53011 ./traces/amptjp-bal.rep
 * yes     19%    5848  0.000110 53405 ./traces/cccp-bal.rep
 * yes     30%    6648  0.000126 52653 ./traces/cp-decl-bal.rep
 * yes     40%    5380  0.000099 54383 ./traces/expr-bal.rep
 * yes      0%   14400  0.000262 54929 ./traces/coalescing-bal.rep
 * yes     38%    4800  0.000108 44556 ./traces/random-bal.rep
 * yes     55%    6000  0.000080 75009 ./traces/binary-bal.rep
10         41%   62295  0.001144 54449

Perf index = 26 (util) + 40 (thru) = 66/100
```

구현 방법

- 이 방법은 말 그대로 "단순 무식하게" ('naive') 메모리 할당 요청이 들어올 때마다 힙 메모리 영역의 크기를 증가하는 방법이다.

- POSIX.1 표준을 준수하는 GNU/Linux 계열 운영 체제에서는 힙 메모리 영역의 크기를 변경하기 위해 `sbrk()` [1] (`'legacy' in POSIX.1-1998`) 또는 `mmap()`[2] 등의 시스템 콜을 이용하는데, 이 중에서 `sbrk()` 함수는 현재 실행 중인 프로세스의 "초기화되지 않은 데이터 영역" (`.bss`)의 시작 위치 변경을 통해 힙 메모리 영역 크기를 변경하는 함수이다.



[1] https://pubs.opengroup.org/onlinepubs/007908775/xsh/brk.html

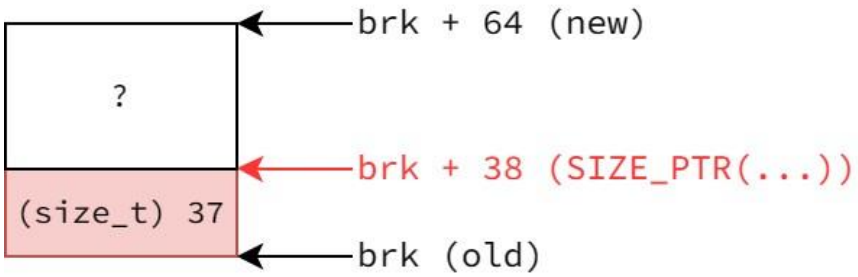
[2] https://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html

- Malloc Lab에서는 sbrk() 시스템 콜을 실제로 사용하는 대신, memlib.c 모듈에서 40 MB (MAX_HEAP)의 정적 배열과 그 배열의 임의의 위치를 가리키는 mem_brk 포인터 변수를 정의하고 mem_sbrk()을 이용하여 sbrk()의 동작을 재현하는 것을 확인할 수 있다.

```
64 /*
65  * Maximum heap size in bytes
66  */
67 #define MAX_HEAP (40*(1<<20))  /* 40 MB */
68
~/malloclab-handout/config.h [utf-8,unix][cpp]
```

```
40 /* single word (4) or double word (8) alignment */
41 #define ALIGNMENT 8
42
43 /* rounds up to the nearest multiple of ALIGNMENT */
44 #define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
45
46
47 #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
48
49 #define SIZE_PTR(p)  ((size_t*)(((char*)(p)) - SIZE_T_SIZE))
50
~/malloclab-handout/mm-naive.c [utf-8,unix][c]
```

- mm-naive.c에서 정의된 상수형 (object-like) 및 함수형 (function-like) 매크로[3]를 살펴보자. 먼저, ALIGN(size)는 비트 연산을 통해 size와 크면서 size와 가장 가까운 ALIGNMENT의 배수를 반환하는 매크로임을 알 수 있다. (예를 들면, size가 31일 때 ALIGN(31)의 값은 32가 될 것이다.)



- 또한, SIZE_PTR(p)는 malloc()으로 할당된 블록의 크기가 저장된 곳의 메모리 주소를 반환한다. 예를 들어, sizeof(size_t)가 26이고 ptr = malloc(37);와 같이 malloc()을 호출했을 때 SIZE_PTR(ptr)의 값은 위 그림과 같이 ptr의 데이터 크기가 저장된 곳의 메모리 주소가 될 것이다.

```
/*
 * mm-naive.c - The fastest, least memory-efficient malloc package.
 *
 * In this naive approach, a block is allocated by simply incrementing
 * the brk pointer.  Blocks are never coalesced or reused.  The size of
 * a block is found at the first aligned word before the block (we need
 * it for realloc).
 *
 * This code is correct and blazingly fast, but very bad usage-wise since
 * it never frees anything.
 */
```

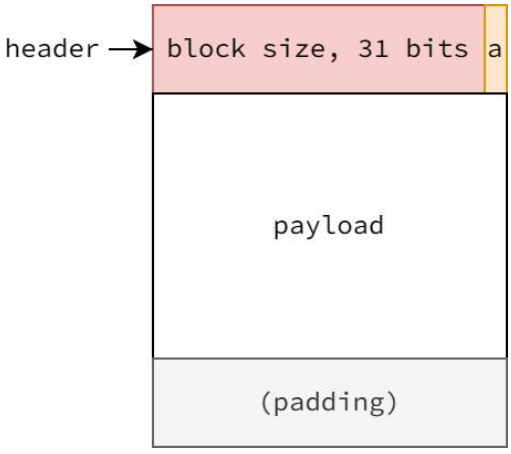
- 결과적으로, 이 방식은 메모리 할당 속도는 매우 빠르겠지만... 블록이 합쳐지거나 재사용되지 않기 때문에 메모리 공간을 비효율적으로 사용한다고 할 수 있다.

[3] https://en.cppreference.com/w/c/preprocessor/replace

Implicit #1

```
~/malloclab-handout $ make && ./mdriver
gcc -Wall -g -DDRIVER   -c -o mm.o mm.c
gcc -Wall -g -DDRIVER -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3791.3 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000001 15917 ./traces/malloc.rep
   yes    28%      17  0.000001 27403 ./traces/malloc-free.rep
   yes    96%      15  0.000001 16820 ./traces/corners.rep
 * yes    86%    1494  0.002974   502 ./traces/perl.rep
 * yes    75%     118  0.000024  4980 ./traces/hostname.rep
 * yes    91%   11913  0.159700    75 ./traces/xterm.rep
 * yes    99%    5694  0.012946   440 ./traces/amptjp-bal.rep
 * yes    99%    5848  0.012105   483 ./traces/cccp-bal.rep
 * yes    99%    6648  0.019745   337 ./traces/cp-decl-bal.rep
 * yes   100%    5380  0.014808   363 ./traces/expr-bal.rep
 * yes    66%   14400  0.000444 32451 ./traces/coalescing-bal.rep
 * yes    93%    4800  0.012891   372 ./traces/random-bal.rep
 * yes    55%    6000  0.045468   132 ./traces/binary-bal.rep
10         86%   62295  0.281105   222

Perf index = 56 (util) + 9 (thru) = 65/100
```

구현 방법



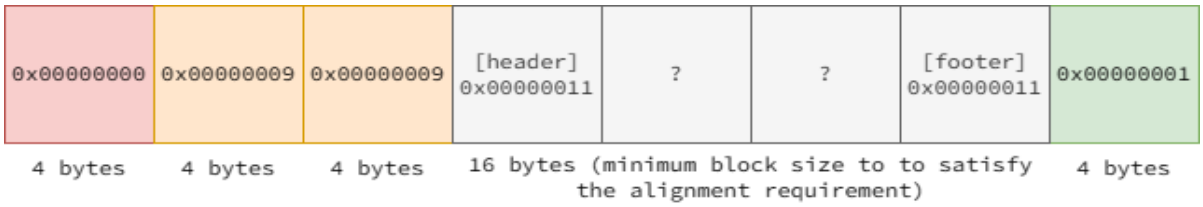
- 간접 리스트 방식은 힙 메모리 영역에서 블록을 할당할 때 첫 번째 워드 (CS:APP3e에서는 4바이트)에 이 블록에 대한 정보를 저장해놓는 방식으로, 이 워드를 헤더 (header)라고 부른다.

- 헤더에는 블록의 실제 데이터를 뜻하는 페이로드 (payload)와 패딩 바이트 (padding)를 포함한 블록 크기, 그리고 이 블록이 할당된 블록임을 나타내는 할당 비트 (allocated bit)가 포함된다. 예를 들어, `ptr = malloc(1);`이라면 `ptr`가 가리키는 블록은 헤더까지 합쳐서 5바이트가 되어야 하지만, 실제로는 블록 사이즈가 8의 배수로 올림되기 때문에 이 블록의 크기는 8바이트 (여기에 헤더와 푸터까지 포함하면 8바이트가 추가되므로, 총 16바이트)이 된다. 또한, 헤더는 블록 크기에 할당 비트를 더한 값이기 때문에 `0x1000 + 0x1 = 0x1001`이 된다.

- 간접 리스트 방식에서 블록을 할당할 때는 프로그래머의 요청에 맞으면서 아직 할당되지 않은 블록의 위치를 효율적으로 찾아야 하는데, 이러한 탐색 방법에는 최초 할당 (first fit), 다음 할당 (next fit), 그리고 최적 할당 (best fit) 방법 등이 있다. 최초 할당 방식은 아직 할당되지 않은 블록들을 처음부터 확인하다가 프로그래머의 요청에 맞는 첫 번째 블록을 선택하여 할당하는 방식이

고, 다음 할당 방식은 최초 할당 방식과 유사하지만 이전에 탐색이 종료되었던 위치에서부터 확인을 시작하는 방식이며, 마지막으로 최적 할당 방식은 아직 할당되지 않은 블록들을 모두 확인해서 프로그래머의 요청에 가장 근접한 크기의 블록을 할당하는 방식이다.

- free()를 이용해 블록을 할당 해제할 때, 힙 메모리 영역을 효율적으로 사용하기 위해서는 그 블록의 이전 블록 또는 다음 블록이 아직 할당되지 않았을 때 두 블록을 합치는 과정이 필요한데, 이것을 블록의 병합 (coalescing) 과정이라고 한다. 블록의 맨 끝에 헤더의 내용을 복사하여 만들어진 푸터 (footer)를 추가하는 경계 태그 (bountary tags) 기법을 이용하면 현재 블록의 이전 또는 다음 블록의 정보를 상수 시간 내에 확인할 수 있다.




```c
/* Private variables */

/*
 * A global pointer variable that points to the first word and the first
 * regular block of the implicit free list.
 */
static char *heap_ptr = NULL, *first_bp = NULL;

/* Private function prototypes */

static void *coalesce(void *bp);
static void *extend_heap(size_t words);
static void *find_fit(size_t adj_size);

static void place(void *bp, size_t adj_size);

/* Public functions */

/*
 * Initialize: return -1 on error, 0 on success.
 */
int mm_init(void) {
    /* 힙 메모리 영역을 초기화한다. */
    if ((heap_ptr = mem_sbrk(4 * WSIZE)) == (void *) -1) return -1;

    PUT(heap_ptr + (0 * WSIZE), 0);                 // 패딩 데이터
    PUT(heap_ptr + (1 * WSIZE), PACK(DSIZE, 1));  // 프롤로그 헤더
    PUT(heap_ptr + (2 * WSIZE), PACK(DSIZE, 1));  // 프롤로그 푸터
    PUT(heap_ptr + (3 * WSIZE), PACK(0, 1));      // 에필로그 헤더

    // 이제 `first_bp`는 프롤로그 푸터를 가리킨다.
    first_bp = heap_ptr + (2 * WSIZE);

#ifdef DEBUG
    dbg_printf("> mm_init(): \n"), mm_checkheap(1);
#endif

    // `CHUNKSIZE`만큼 힙 메모리 영역을 확장한다.
    return (extend_heap(CHUNKSIZE / WSIZE) != NULL) - 1;
}
```

- mm_init() 함수는 힙 메모리 영역을 초기화하는 함수이다. **힙 메모리 영역을 초기화할 때는 4바이트의 패딩 바이트, 헤더와 푸터로만 이루어진 8바이트의 프롤로그 블록 (prologue block), 그리고 크기가 0으로 설정된 헤더만으로 이루어진 에필로그 블록 (epilogue block)를 생성한다.** 프롤로그 블록과 에필로그 블록을 생성하는 이유는 블록 병합 과정을 편리하게 하기 위함이다. mm_init() 함수의 정의를 살펴보면 PUT()이라는 함수형 매크로를 이용하는 것을 알 수 있는데, 이 매크로는 주어진 메모리 주소에서부터 시작하는 4바이트 공간 (sizeof(unsigned int) = 4라고 가정함)에 값을 쓰는 기능을 수행한다. 또다른 함수형 매크로인 PACK()은 블록의 크기 정보와 할당 비트를 OR 연산으로 합쳐 하나의 4바이트 값으로 만든다.

- extend_heap() 함수는 주어진 워드 개수에 맞춰 힙 메모리 영역을 확장하는 함수이다. 힙 메모리 영역 확장이 끝난 후 생성된 새로운 블록의 헤더와 푸터는 프롤로그 블록과 마찬가지로 PUT() 매크로와 PACK() 매크로를 이용해 생성한다.

```c
/*
 * Extend the heap with a new free block.
 */
static void *extend_heap(size_t words) {
    /*
     * 정렬 유지를 위해 블록 크기를 항상 짝수 개의
     * 워드로 설정한다.
     */
    size_t size = (words + (words & 1)) * WSIZE;

    char *bp = mem_sbrk(size);

    if (bp == (char *) -1) return NULL;

    PUT(HDRP(bp), PACK(size, 0));          // 블록 헤더
    PUT(FTRP(bp), PACK(size, 0));          // 블록 푸터
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));  // 에필로그 푸터

#ifdef DEBUG
    dbg_printf("> extend_heap(): \n"), mm_checkheap(1);
#endif

    return coalesce(bp);
}
```

```c
/*
 * Coalesce this block with adjacent free block(s).
 */
static void *coalesce(void *bp) {
    // 이전 블록과 다음 블록의 할당 비트를 확인한다.
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));

    if (prev_alloc && next_alloc) return bp;

    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && !next_alloc) {
        /*
         * Case 2: 이전 블록은 비가용, 다음 블록은 가용인 경우...?
         */

        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) {
        /*
         * Case 3: 이전 블록은 가용, 다음 블록은 비가용인 경우...?
         */

        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 {
        /*
         * Case 4: 이전 블록과 다음 블록이 둘 다 가용인 경우...?
         */

        size += GET_SIZE(HDRP(PREV_BLKP(bp)))
            + GET_SIZE(FTRP(NEXT_BLKP(bp)));

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

        bp = PREV_BLKP(bp);
    }

#ifdef DEBUG
    dbg_printf("> coalesce(): \n"), mm_checkheap(1);
#endif

    return bp;
}
```

- coalesce() 함수는 bp가 가리키는 블록의 이전 블록이나 다음 블록의 할당 비트를 GET_ALLOC() 매크로를 통해 확인하고, 이전 블록이나 다음 블록이 가용 블록일 경우 그 블록과 bp가 가리키는 블록을 하나로 합치는 기능을 수행한다. **이때 중요한 것은 하나로 합친 블록의 크기를 다시 설정해야 한다는 것과 bp가 가리키는 블록을 이전 블록과 합쳤을 경우 bp가 이전 블록을 가리키게 해야 한다는 것이다.**

```c
/*
 * Allocates `size` bytes of uninitialized storage.
 *
 * If allocation succeeds, returns a pointer that is suitably aligned
 * for any object type with fundamental alignment.
 *
 * If `size` is zero, the behavior of malloc is implementation-defined.
 *
 * For example, a null pointer may be returned. Alternatively,
 * a non-null pointer may be returned; but such a pointer should not
 * be dereferenced, and should be passed to `free()` to avoid memory leaks.
 */
void *malloc(size_t size) {
    if (size == 0) return NULL;

    char *bp = NULL;

    size_t adj_size = (size > DSIZE)
        ? (DSIZE * ((size + DSIZE + (DSIZE - 1)) / DSIZE))
        : (DSIZE << 1); /* 블록의 최소 크기 */

#ifdef DEBUG
    dbg_printf("> malloc() #1: \n"), mm_checkheap(1);
#endif

    if ((bp = find_fit(adj_size)) != NULL) {
        // 적절한 가용 블록을 찾았으므로, 이 블록을 할당한다.
        place(bp, adj_size);

#ifdef DEBUG
        dbg_printf("> malloc() #2.1: \n"), mm_checkheap(1);
#endif

        return bp;
    } else {
        // 가용 블록이 없으므로, 힙 메모리 영역을 확장한다.
        size_t ext_size = MAX(adj_size, CHUNKSIZE);

        if ((bp = extend_heap(ext_size / WSIZE)) != NULL) {
            place(bp, adj_size);

#ifdef DEBUG
            dbg_printf("> malloc() #2.2: \n"), mm_checkheap(1);
#endif

            return bp;
        }

        return NULL;
    }
}
```

- `malloc()` 함수는 힙 메모리 영역에 크기가 `size`인 블록을 생성하고, 그 블록의 메모리 주소를 반환하는 함수이다. 블록을 생성할 때는 8바이트의 메모리 공간 정렬을 유지하기 위해 최소 16바이트 이상의 블록을 생성하도록 한다. 만약 `size`가 헤더와 푸터를 합친 크기인 8바이트보다 작으면 `adj_size`는 16바이트가 되고, `size`가 8바이트보다 크거나 같으면 `size`보다 크면서 메모리 공간 정렬을 유지하기 위한 8의 배수가 `adj_size`의 값이 된다. 따라서 `malloc()`을 호출하면 실제로는 크기가 `adj_size`인 블록이 반환될 것이다.

```c
/*
 * Perform a first-fit search of the implicit free list.
 */
static void *find_fit(size_t adj_size) {
    // 간접 리스트의 첫 번째 가용 블록부터 차례대로 확인한다.
    for (void *bp = first_bp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp))
        if (GET_SIZE(HDRP(bp)) >= adj_size && !GET_ALLOC(HDRP(bp))) {
#ifdef DEBUG
            dbg_printf(
                "> find_fit(): Found:\n"
                "| [BLOCK.HDR] %u: %u\n\n",
                GET_SIZE(HDRP(bp)),
                GET_ALLOC(HDRP(bp))
            );
#endif
            return bp;
        }

#ifdef DEBUG
    dbg_printf("> find_fit(): Not found...\n\n");
#endif

    return NULL;
}
```

- find_fit() 함수는 간접 리스트를 첫 번째 가용 블록부터 순서대로 탐색하면서, adj_size보다 크거나 같은 첫 번째 가용 블록을 찾아 그 블록의 메모리 주소를 반환하는 최초 할당 방식을 이용한다.

- place() 함수는 크기가 adj_size인 블록을 힙 메모리 영역에 실제로 할당해주는 함수이다. **여기서 주목해야 할 부분은 bp가 가리키는 블록의 크기가 adj_size보다 크면서 그 차이가 최소 블록 크기보다 클 경우, 이 블록을 2개의 블록으로 나눠줘야 한다는 것이다.** 이렇게 되면 첫 번째 블록은 크기 adj_size의 비가용 블록이 되고, 두 번째 블록은 크기가 remainder인 가용 블록이 된다.

```c
/*
 * Place the requested block at the beginning of the free block,
 * splitting only if the size of the remainder would equal or
 * exceed the minimum block size.
 */
static void place(void *bp, size_t adj_size) {
    size_t size = GET_SIZE(HDRP(bp)), remainder = size - adj_size;

#ifdef DEBUG
    dbg_printf("> place() #1: \n"), mm_checkheap(1);
#endif

    // 가용 블록의 크기와 할당 요청 크기를 비교한다.
    if (remainder < (DSIZE << 1)) {
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    } else {
        PUT(HDRP(bp), PACK(adj_size, 1));
        PUT(FTRP(bp), PACK(adj_size, 1));

        bp = NEXT_BLKP(bp);

        // 이 블록은 가용 블록이 된다.
        PUT(HDRP(bp), PACK(remainder, 0));
        PUT(FTRP(bp), PACK(remainder, 0));
    }

#ifdef DEBUG
    dbg_printf("> place() #2: \n"), mm_checkheap(1);
#endif
}
```

```c
/*
 * Check the heap for correctness.
 */
void mm_checkheap(int verbose) {
    void *bp = first_bp;

    // 패딩 바이트를 검사한다.
    assert(GET(heap_ptr + (0 * WSIZE)) == 0);

    // 프롤로그 헤더를 검사한다.
    assert(GET_SIZE(HDRP(first_bp)) == DSIZE);
    assert(GET_ALLOC(HDRP(first_bp)));

    if (verbose)
        dbg_printf(
            "| [PADDING] %u |\n"
            "| [PROLOGUE.HDR] %u: %u "
            "| [PROLOGUE.FTR] %u: %u |\n",
            GET(heap_ptr + (0 * WSIZE)),
            GET_SIZE(HDRP(first_bp)), GET_ALLOC(HDRP(first_bp)),
            GET_SIZE(FTRP(first_bp)), GET_ALLOC(HDRP(first_bp))
        );

    // 각 블록을 차례대로 확인한다.
    for (; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (verbose)
            dbg_printf(
                "| [BLOCK.HDR] %u: %u "
                "| [BLOCK.DATA] %u bytes "
                "| [BLOCK.FTR] %u: %u |\n",
                GET_SIZE(HDRP(bp)), GET_ALLOC(HDRP(bp)),
                GET_SIZE(HDRP(bp)) - DSIZE,
                GET_SIZE(FTRP(bp)), GET_ALLOC(FTRP(bp))
            );

        assert(GET(HDRP(bp)) == GET(FTRP(bp)));
    }

    // 에필로그 헤더를 검사한다.
    assert(GET_SIZE(HDRP(bp)) == 0);
    assert(GET_ALLOC(HDRP(bp)));

    if (verbose)
        dbg_printf(
            "| [EPILOGUE.HDR] %u: %u |\n\n",
            GET_SIZE(HDRP(bp)), GET_ALLOC(HDRP(bp))
        );
}
```

- `mm_checkheap()` 함수는 효율적인 디버깅을 위해 힙 메모리 영역 전체를 `assert()` 매크로[4]를 이용해 검사하고, `DEBUG` 매크로가 정의되어 있다면 각 블록의 구조를 출력하는 기능을 수행한다. `assert()` 매크로는 주어진 조건을 만족하지 않는다면 실행 중인 프로세스에 `SIGABRT` 시그널을 보내 프로세스를 즉시 종료시키는데, 이러한 특성으로 인해 C 프로그램의 단위 테스트나 디버깅에 자주 사용된다.

---

[4] https://en.cppreference.com/w/c/error/assert

Implicit #2

```
~/malloclab-handout $ make implicit && make && ./mdriver
rm -f mm.c mm.o; ln -s mm-implicit.c mm.c
gcc -Wall -g -DDRIVER   -c -o mm.o mm.c
gcc -Wall -g -DDRIVER -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3791.3 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000001 18976 ./traces/malloc.rep
   yes    28%      17  0.000001 18763 ./traces/malloc-free.rep
   yes    96%      15  0.000001 18862 ./traces/corners.rep
 * yes    81%    1494  0.000183  8185 ./traces/perl.rep
 * yes    75%     118  0.000005 22489 ./traces/hostname.rep
 * yes    91%   11913  0.002180  5464 ./traces/xterm.rep
 * yes    91%    5694  0.004005  1422 ./traces/amptjp-bal.rep
 * yes    92%    5848  0.002706  2161 ./traces/cccp-bal.rep
 * yes    95%    6648  0.007382   901 ./traces/cp-decl-bal.rep
 * yes    97%    5380  0.007697   699 ./traces/expr-bal.rep
 * yes    66%   14400  0.000606 23773 ./traces/coalescing-bal.rep
 * yes    91%    4800  0.008422   570 ./traces/random-bal.rep
 * yes    55%    6000  0.006063   990 ./traces/binary-bal.rep
10        83%   62295  0.039249  1587

Perf index = 54 (util) + 40 (thru) = 94/100
```

구현 방법

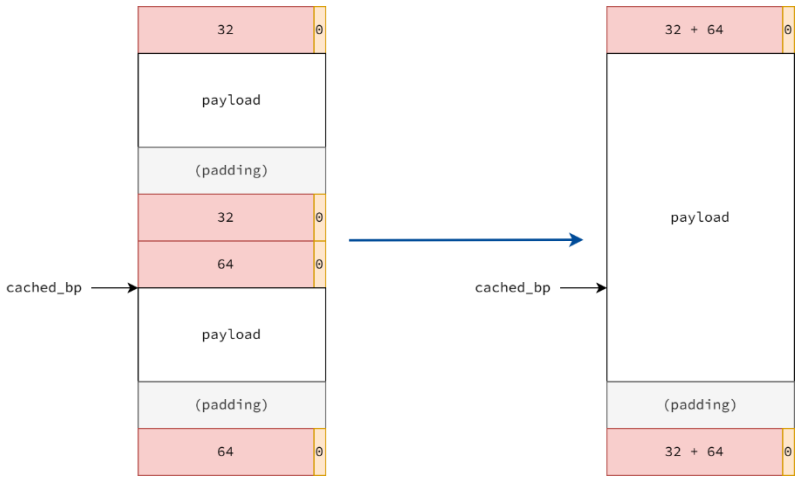
```
    /*
     * 이전 블록의 크기가 `(DSIZE << 1)`보다 작다는 것은
     * 그 블록이 프롤로그 블록임을 뜻한다.
     */
    if (GET_SIZE(PREV_BLKP(bp)) < (DSIZE << 1))
        first_bp = bp;

#ifdef DEBUG
    dbg_printf("> coalesce(): \n"), mm_checkheap(1);
#endif

    return bp;
}
```

- 간접 리스트 할당자의 성능을 개선해보자. 첫 번째로 할 일은 `first_bp`가 항상 프롤로그 푸터만을 가리키게 하는 대신, 간접 리스트의 첫 번째 일반 블록(regular block)을 가리키게 하는 것이다. 일반 블록의 최소 크기는 `(DSIZE << 1)`, 곧 16바이트이므로, 어떤 블록을 선택했을 때 이 블록의 이전 블록의 크기가 16바이트보다 작다는 것은 곧 우리가 선택한 블록이 프롤로그 블록 다음에 위치한 첫 번째 일반 블록임을 뜻한다.

- 두 번째로 할 일은 가용 블록을 찾기 위해 최초 할당 방식 대신 다음 할당 방식을 이용하는 것이다. 다음 할당 방식을 구현하기 위해 이전 탐색 시의 탐색 종료 위치를 저장할 `cached_bp` 포인터 변수를 선언 및 정의한다.

```
/* Private variables */

/*
 * A global pointer variable that points to the first word, the first
 * regular block, and the cached regular block of the implicit free list.
 */
static char *heap_ptr = NULL, *first_bp = NULL, *cached_bp = NULL;
```

```c
/*
 * Perform a (first / next / best)-fit search of the implicit free list.
 */
static void *find_fit(size_t adj_size) {
    // 이전 탐색 종료 위치에 해당하는 블록부터 차례대로 확인한다.
    for (void *bp = cached_bp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp))
        if (GET_SIZE(HDRP(bp)) >= adj_size && !GET_ALLOC(HDRP(bp))) {
            dbg_printf(
                "> find_fit() #1: Found:\n"
                "| [BLOCK.HDR] %u: %u\n\n",
                GET_SIZE(HDRP(bp)),
                GET_ALLOC(HDRP(bp))
            );

            return (cached_bp = bp);
        }

    // 이제 첫 번째 가용 블록부터 이전 탐색 종료 위치 직전까지 확인한다.
    for (void *bp = first_bp; bp < (void *) cached_bp; bp = NEXT_BLKP(bp))
        if (GET_SIZE(HDRP(bp)) >= adj_size && !GET_ALLOC(HDRP(bp))) {
            dbg_printf(
                "> find_fit() #2: Found:\n"
                "| [BLOCK.HDR] %u: %u\n\n",
                GET_SIZE(HDRP(bp)),
                GET_ALLOC(HDRP(bp))
            );

            return (cached_bp = bp);
        }

    dbg_printf("> find_fit(): Not found...\n\n");

    return NULL;
}
```



- find_fit() 함수를 구현할 때 유의해야 할 점은 그림과 같이 cached_bp가 블록 병합 과정으로 인해 블록의 페이로드 시작 위치가 아닌 페이로드 내부의 임의의 위치를 가리키게 될 수도 있다는 것이다. 그렇기 때문에 블록 병합 과정이 끝난 후에 cached_bp를 확인 및 재설정해주어야 한다.

```c
/*
 * 이전 탐색 종료 위치가 블록의 페이로드 시작 위치가
 * 아닌 페이로드 내부의 다른 위치를 가리키고 있다면,
 * 그 위치를 재설정한다.
 */
if ((cached_bp > (char *) bp)
    && (cached_bp < NEXT_BLKP(bp))) cached_bp = bp;
```

Explicit #1

```
~/malloclab-handout $ make explicit && make && ./mdriver
rm -f mm.c mm.o; ln -s mm-explicit.c mm.c
gcc -Wall -g -DDRIVER   -c -o mm.o mm.c
gcc -Wall -g -DDRIVER -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 1000.0 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000002  5136 ./traces/malloc.rep
   yes    28%      17  0.000003  5402 ./traces/malloc-free.rep
   yes    96%      15  0.000003  4817 ./traces/corners.rep
 * yes    81%    1494  0.000329  4541 ./traces/perl.rep
 * yes    75%     118  0.000023  5195 ./traces/hostname.rep
 * yes    89%   11913  0.002432  4898 ./traces/xterm.rep
 * yes    89%    5694  0.001796  3170 ./traces/amptjp-bal.rep
 * yes    92%    5848  0.001470  3978 ./traces/cccp-bal.rep
 * yes    94%    6648  0.002447  2717 ./traces/cp-decl-bal.rep
 * yes    96%    5380  0.002037  2641 ./traces/expr-bal.rep
 * yes    66%   14400  0.002481  5804 ./traces/coalescing-bal.rep
 * yes    87%    4800  0.003051  1573 ./traces/random-bal.rep
 * yes    55%    6000  0.003534  1698 ./traces/binary-bal.rep
10         82%   62295  0.019600  3178

Perf index = 53 (util) + 40 (thru) = 93/100
~/malloclab-handout $
```

구현 방법

- 직접 리스트 방식에서는 가용 블록에 이전 가용 블록과 다음 가용 블록의 메모리 주소를 저장해놓고 가용 블록 탐색 시에 사용하는 "이중 연결 리스트" (doubly linked list)의 형태로 가용 블록을 관리한다. 직접 리스트 방식에서 `mm_init()` 함수의 구현 방법은 간접 리스트 방식에서의 `mm_init()` 함수 구현 방법과 유사하지만, 패딩 데이터 앞에 첫 번째 가용 블록과 마지막 가용 블록의 메모리 주소를 저장하기 위한 16바이트의 공간을 추가적으로 할당한다는 점이 다르다고 할 수 있다.

```c
/* Public functions */

/*
 * Initialize: return -1 on error, 0 on success.
 */
int mm_init(void) {
    int incr = WSIZE + (OVERHEAD + (DSIZE << 1)) + HDRSIZE;

    /* 힙 메모리 영역을 초기화한다. */
    if ((heap_ptr = mem_sbrk(incr)) == (void *) -1) return -1;

    PUT8(heap_ptr,         (unsigned long) NULL);     // 첫 번째 가용 블록
    PUT8(heap_ptr + DSIZE, (unsigned long) NULL);     // 마지막 가용 블록

    PUT(heap_ptr + (DSIZE << 1), 0);                  // 패딩 데이터

    heap_ptr += ((DSIZE << 1) + WSIZE);

    PUT(heap_ptr, PACK(OVERHEAD, 1));                 // 프롤로그 헤더
    PUT(heap_ptr + HDRSIZE, PACK(OVERHEAD, 1));       // 프롤로그 푸터
    PUT(heap_ptr + (HDRSIZE + FTRSIZE), PACK(0, 1));  // 에필로그 헤더

    heap_ptr -= ((DSIZE << 1) + WSIZE);

    dbg_printf("> mm_init(): \n"); mm_checkheap(1);

    // `CHUNKSIZE`만큼 힙 메모리 영역을 확장한다.
    return (extend_heap(CHUNKSIZE / WSIZE) != NULL) - 1;
}
```

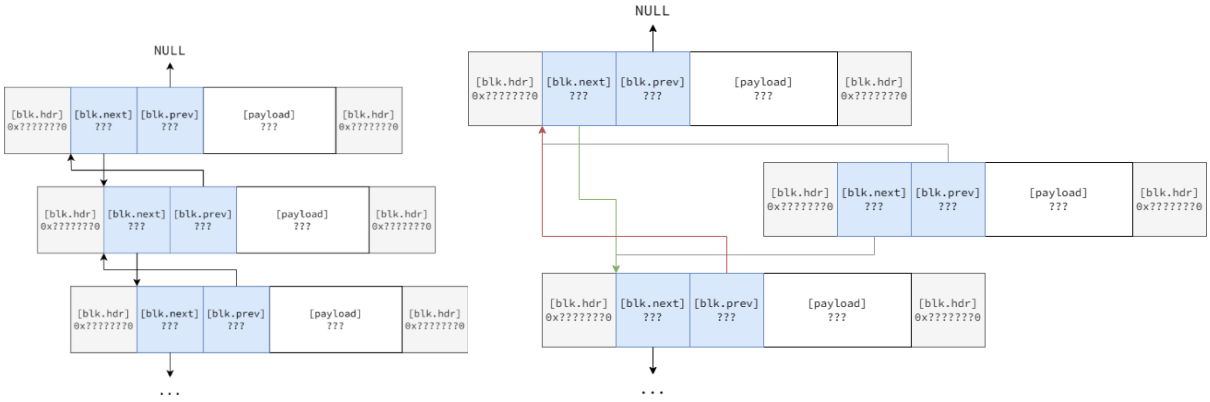
- 첫 번째 가용 블록과 마지막 가용 블록의 메모리 주소를 저장할 때는 64비트 GNU/Linux 운영 체제의 `sizeof(void *)`에 맞춰서 각각 8바이트의 공간을 할당해주었으며, 그 외 나머지 부분은 간접 리스트에서의 `mm_init()`과 동일하게 구현하였다.

```
> insert_into_list() #4:
| [ROOT.HEAD] <0x56262c552560> | [ROOT.TAIL] <(nil)> | [PADDING] 0 |
| [PROLG.HDR <0x56262c552558>] 8: 1 | [PROLG.FTR] 8: 1 |
| [BLOCK.HDR <0x56262c552558>] 8: 1 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] (nil) | [BLOCK.FTR] 8: 1 |
| [BLOCK.HDR <0x56262c552560>] 40: 0 | [BLOCK.NEXT] 0x56262c5525f8 | [BLOCK.PREV] (nil) | [BLOCK.FTR] 40: 0 |
| [BLOCK.HDR <0x56262c552588>] 40: 1 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] (nil) | [BLOCK.FTR] 40: 1 |
| [BLOCK.HDR <0x56262c5525b0>] 72: 1 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] (nil) | [BLOCK.FTR] 72: 1 |
| [BLOCK.HDR <0x56262c5525f8>] 3944: 0 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] 0x56262c552560 | [BLOCK.FTR] 3944: 0 |
=========================================
| [FREE.HDR <0x56262c552560>] 40: 0 | [FREE.NEXT] 0x56262c5525f8 | [FREE.PREV] (nil) | [FREE.FTR] 40: 0 |
| [FREE.HDR <0x56262c5525f8>] 3944: 0 | [FREE.NEXT] (nil) | [FREE.PREV] 0x56262c552560 | [FREE.FTR] 3944: 0 |
=========================================
| [EPILG.HDR <0x56262c553560>] 0: 1 |

> coalesce() #2:
| [ROOT.HEAD] <0x56262c552560> | [ROOT.TAIL] <(nil)> | [PADDING] 0 |
| [PROLG.HDR <0x56262c552558>] 8: 1 | [PROLG.FTR] 8: 1 |
| [BLOCK.HDR <0x56262c552558>] 8: 1 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] (nil) | [BLOCK.FTR] 8: 1 |
| [BLOCK.HDR <0x56262c552560>] 40: 0 | [BLOCK.NEXT] 0x56262c5525f8 | [BLOCK.PREV] (nil) | [BLOCK.FTR] 40: 0 |
| [BLOCK.HDR <0x56262c552588>] 40: 1 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] (nil) | [BLOCK.FTR] 40: 1 |
| [BLOCK.HDR <0x56262c5525b0>] 72: 1 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] (nil) | [BLOCK.FTR] 72: 1 |
| [BLOCK.HDR <0x56262c5525f8>] 3944: 0 | [BLOCK.NEXT] (nil) | [BLOCK.PREV] 0x56262c552560 | [BLOCK.FTR] 3944: 0 |
=============================================
| [FREE.HDR <0x56262c552560>] 40: 0 | [FREE.NEXT] 0x56262c5525f8 | [FREE.PREV] (nil) | [FREE.FTR] 40: 0 |
| [FREE.HDR <0x56262c5525f8>] 3944: 0 | [FREE.NEXT] (nil) | [FREE.PREV] 0x56262c552560 | [FREE.FTR] 3944: 0 |
=========================================
| [EPILG.HDR <0x56262c553560>] 0: 1 |
```

- `mm_checkheap()`에는 각각의 가용 블록이 제대로 연결되어 있는지 확인하기 위해 루트 블록에 저장된 첫 번째 가용 블록의 메모리 주소에서부터 시작해서 모든 가용 블록을 출력하는 코드를 추가하였다.

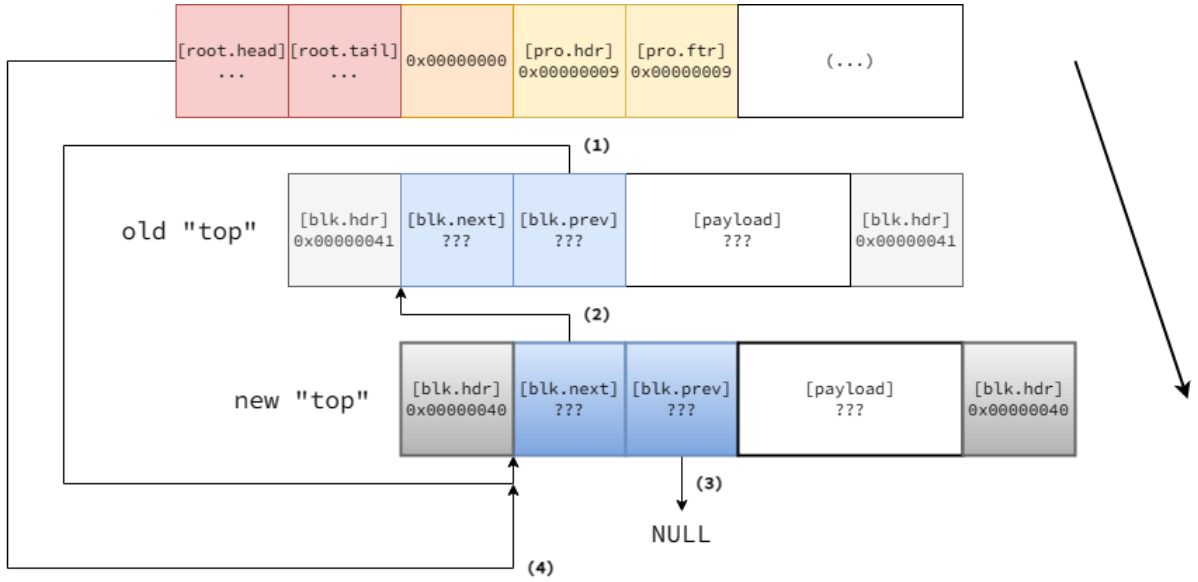
```c
    // 모든 블록의 헤더와 푸터를 검사한다.
    for (; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
#if 1
        if (verbose) print_block(bp, NULL, 0);
#endif
        assert(GET(HDRP(bp)) == GET(FTRP(bp)));
    }

    void *fbp = (void *) GET8(heap_ptr);

    if (fbp != NULL) {
        dbg_printf("=========================================\n");

        // 가용 블록의 헤더와 푸터, 그리고 링크 포인터를 검사한다.
        for (; fbp != NULL; fbp = NEXT_FREE_BLKP(fbp)) {
            assert((fbp > mem_heap_lo()) && (fbp < mem_heap_hi()));

            if (verbose) print_block(fbp, "FREE", 0);
        }
```

- coalesce()와 place() 함수에서는 블록이 하나로 합쳐지거나 둘로 나눠지는 상황이 발생할 수 있는데, 이때 연결 리스트에서 특정 블록을 제거하거나 연결 리스트의 맨 앞에 새로운 가용 블록을 추가하는 함수가 필요할 것이라고 판단하였다.



- remove_from_list()는 이중 연결 리스트에서 주어진 블록과 이전 가용 블록 사이의 연결, 그리고 주어진 블록과 다음 가용 블록 사이의 연결을 끊고, 이전 가용 블록과 다음 가용 블록을 서로 연결해주는 함수이다. 이 함수를 구현할 때 주의할 점은 NEXT_FREEP() 매크로가 다음 가용 블록의 메모리 주소가 아닌, 블록에서 다음 가용 블록의 메모리 주소가 저장되는 블록 내의 공간을 가리킨다는 것이다. **즉, 다음 가용 블록의 메모리 주소를 얻으려면 NEXT_FREE_BLKP() 매크로를 이용하거나 (void *) GET8(NEXT_FREEP(bp))와 같이 NEXT_FREEP(bp)에 저장된 값을 포인터 형태로 캐스팅해야 한다.** (이전 가용 블록의 메모리 주소도 같은 방법으로 얻을 수 있다!)

```c
/*
 * Remove the block from the free list.
 */
static void remove_from_list(void *bp) {
    if (bp == NULL) return;

    dbg_printf("> remove_from_list() #1: \n"); print_block(bp, NULL, 1);

    void *next_fbp = NEXT_FREE_BLKP(bp), *prev_fbp = PREV_FREE_BLKP(bp);

    if (prev_fbp != NULL && !GET_ALLOC(HDRP(prev_fbp))) {
        // `prev_fbp`의 다음 링크를 변경한다.
        PUT8(NEXT_FREEP(prev_fbp), (unsigned long) next_fbp);

        dbg_printf("> remove_from_list() #2.1: \n"); print_block(prev_fbp, NULL, 1);
    }

    if (next_fbp != NULL && !GET_ALLOC(HDRP(next_fbp))) {
        // `next_fbp`의 이전 링크를 변경한다.
        PUT8(PREV_FREEP(next_fbp), (unsigned long) prev_fbp);

        dbg_printf("> remove_from_list() #2.2: \n"); print_block(next_fbp, NULL, 1);
    }

    // `bp`가 첫 번째 가용 블록일 경우...?
    if (bp == (void *) GET8(heap_ptr))
        PUT8(heap_ptr, (unsigned long) next_fbp);

    dbg_printf("> remove_from_list() #3: \n"); mm_checkheap(1);
}
```



- insert_into_list() 함수는 주어진 블록을 이중 연결 리스트의 맨 앞에 추가하는 함수인데, 위 그림과 같이 총 4개의 연산을 거친다. 첫 번째 연산은 이중 연결 리스트에서 현재 맨 앞에 위치한 fbp의 이전 링크가 bp를 가리키게 하고, 그 다음에는 bp의 다음 링크가 fbp를 가리키게 하며, bp가 이중 연결 리스트의 맨 앞에 위치하기 때문에 bp의 이전 링크를 NULL로 설정해주고, 마지막으로 루트 블록에 bp를 저장하여 bp가 첫 번째 가용 블록이 되도록 한다.

```c
/*
 * Insert the freed block at the beginning of the free list.
 */
static void insert_into_list(void *bp) {
    if (bp == NULL || GET_ALLOC(FTRP(bp))) return;

    void *fbp = (void *) GET8(heap_ptr);

    dbg_printf("> insert_into_list() #1: \n"); mm_checkheap(1);

    if (bp != fbp) {
        dbg_printf("> insert_into_list() #2.1: \n"); mm_checkheap(1);

        // `fbp`의 이전 링크를 변경한다.
        if (fbp != NULL) {
            PUT8(PREV_FREEP(fbp), (unsigned long) bp);

            dbg_printf("> insert_into_list() #2.2: \n"); print_block(fbp, "FREE", 1);
        }

        // `bp`의 다음 링크를 변경한다.
        PUT8(NEXT_FREEP(bp), (unsigned long) fbp);

        dbg_printf("> insert_into_list() #2.3: \n"); print_block(bp, NULL, 1);
    }

    // `bp`의 이전 링크를 설정한다.
    PUT8(PREV_FREEP(bp), (unsigned long) NULL);

    dbg_printf("> insert_into_list() #3: \n"); print_block(bp, NULL, 1);

    // 이제 `bp`는 첫 번째 가용 블록이 된다.
    PUT8(heap_ptr, (unsigned long) bp);

    dbg_printf("> insert_into_list() #4: \n"); mm_checkheap(1);
}
```

```c
/*
 * Coalesce this block with adjacent free block(s).
 */
static void *coalesce(void *bp) {
    dbg_printf("> coalesce() #1: \n"); mm_checkheap(1);

    // 이전 블록과 다음 블록의 할당 비트를 확인한다.
    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) {
        /*
         * Case 2: 이전 블록은 비가용, 다음 블록은 가용인 경우...?
         */

        remove_from_list(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) {
        /*
         * Case 3: 이전 블록은 가용, 다음 블록은 비가용인 경우...?
         */

        remove_from_list(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 if (!prev_alloc && !next_alloc) {
        /*
         * Case 4: 이전 블록과 다음 블록이 둘 다 가용인 경우...?
         */

        remove_from_list(NEXT_BLKP(bp)), remove_from_list(PREV_BLKP(bp));

        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));

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

        bp = PREV_BLKP(bp);
    }

    insert_into_list(bp);

    dbg_printf("> coalesce() #2: \n"); mm_checkheap(1);

    return bp;
}
```

- `coalesce()` 함수의 구현부는 간접 리스트 방식에서의 구현부와 거의 똑같지만, 블록이 병합될 때 이전 블록 또는 다음 블록을 이중 연결 리스트에서 제거하고 병합이 완료된 블록을 다시 연결 리스트에 추가하기 위해 `remove_from_list()`와 `insert_into_list()` 함수를 이용하는 부분이 추가되어 있다.

```c
/*
 * Place the requested block at the beginning of the free block,
 * splitting only if the size of the remainder would equal or
 * exceed the minimum block size.
 */
static void place(void *bp, size_t adj_size) {
    size_t size = GET_SIZE(HDRP(bp)), remainder = size - adj_size;

    dbg_printf("> place() #1: \n"); mm_checkheap(1);

    // 비가용 블록은 가용 리스트에서 제거한다.
    remove_from_list(bp);

    // 가용 블록의 크기와 할당 요청 크기를 비교한다.
    if (remainder < (OVERHEAD + (DSIZE << 1))) {
        dbg_printf("> place() #2.1: \n"); print_block(bp, NULL, 1);

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

        dbg_printf("> place() #2.2: \n"); print_block(bp, NULL, 1);
    } else {
        dbg_printf("> place() #2.3: \n"); print_block(bp, NULL, 1);

        PUT(HDRP(bp), PACK(adj_size, 1));
        PUT(FTRP(bp), PACK(adj_size, 1));

        dbg_printf("> place() #2.4: \n"); print_block(bp, NULL, 1);

        bp = NEXT_BLKP(bp);

        // 이 블록은 가용 블록이 된다.
        PUT(HDRP(bp), PACK(remainder, 0));
        PUT(FTRP(bp), PACK(remainder, 0));

        dbg_printf("> place() #2.5: \n"); print_block(bp, NULL, 1);

        // 가용 리스트에 블록을 삽입한다.
        insert_into_list(bp);
    }

    dbg_printf("> place() #3: \n"); mm_checkheap(1);
}
```

- place() 함수는 가용 블록의 크기와 할당 요청 크기의 차인 remainder와 최소 블록 크기를 비교하는 부분에서 최소 블록 크기가 (OVERHEAD + (DSIZE << 1)) (헤더 크기와 푸터 크기의 합이 OVERHEAD, 그리고 가용 블록의 이전 링크 포인터와 다음 링크 포인터의 크기가 각각 sizeof(void *), 즉 8바이트이므로…)로 변경된 것, 그리고 블록이 분할되었을 때 연결 리스트를 업데이트하기 위해 remove_from_list()와 insert_into_list() 함수를 이용하는 부분이 추가된 것을 제외하면 나머지 부분은 간접 리스트의 구현부와 동일하다.

```c
/*
 * Perform a (first / next / best)-fit search of the explicit free list.
 */
static void *find_fit(size_t adj_size) {
    void *fbp = (void *) GET8(heap_ptr);

    // 직접 리스트의 첫 번째 가용 블록부터 차례대로 확인한다.
    for (; fbp != NULL && !GET_ALLOC(HDRP(fbp)); fbp = NEXT_FREE_BLKP(fbp)) {
        if ((GET_SIZE(HDRP(fbp)) >= adj_size)) {
            dbg_printf("> find_fit(): Found:\n"); print_block(fbp, NULL, 1);

            return fbp;
        }
    }

    dbg_printf("> find_fit(): Not found...\n\n");

    return NULL;
}
```

- find_fit() 함수는 모든 블록을 확인하는 대신, 가용 블록만을 차례대로 확인하는 최초 할당 방식으로 구현하였다.

- malloc()은 직접 리스트에서의 최소 블록 크기인 (OVERHEAD + (DSIZE << 1))에 맞춰 adj_size를 조정한다는 것을 제외하면 나머지 구현부는 간접 리스트에서의 malloc()과 동일하다.

```c
/*
 * Allocates `size` bytes of uninitialized storage.
 *
 * If allocation succeeds, returns a pointer that is suitably aligned
 * for any object type with fundamental alignment.
 *
 * If `size` is zero, the behavior of `malloc()` is implementation-defined.
 *
 * For example, a null pointer may be returned. Alternatively,
 * a non-null pointer may be returned; but such a pointer should not
 * be dereferenced, and should be passed to `free()` to avoid memory leaks.
 */
void *malloc(size_t size) {
    if (size == (size_t) 0) return NULL;

    char *bp = NULL;

    size_t adj_size = (size > (DSIZE << 1))
        ? DSIZE * ((size + OVERHEAD + (DSIZE - 1)) / DSIZE)
        : (OVERHEAD + (DSIZE << 1));
```