

시스템 프로그래밍

실습 과제(11)(malloc lab - implicit)

01분반

malloc lab

(implicit)

mm-naive 필히 설명

- naive에 사용된 모든 함수 명을 쓰고 함수에 관련된 설명첨부
- 소스코드는 첨부하지 않아도 됨
- macro 설명 포함

mm-implicit 구현 설명

실습 진행

- malloc, realloc, free를 구현하되 높은 성능을 갖도록 구현한다.
- 구현하는 알고리즘에 따라 다른 성능을 갖는다.
- 성능 측정 및 테스트는 테스트 프로그램을 이용하여 확인할 수 있다.
- malloclab-writeup.pdf 문서를 꼭 읽어볼 것.

요구 사항

- malloc, realloc, free 구현
- calloc 제외

malloclab 구성

- mm_implicit.c : implicit list 방식으로 malloc, free, realloc 구현
- mm_explicit.c : explicit list 방식으로 malloc, free, realloc 구현
- mm_seglist.c : segregated free list 방식으로 malloc, free, realloc 구현
- mm_naive : 제공된 코드로 malloc과 realloc 이 구현되어 있음
- memlib.c : dynamic memory allocation을 위한 가상 메모리 시스템
- mdriver.c : malloclab을 테스트할 프로그램
- ./traces : malloclab 테스트 케이스가 있는 디렉토리

make 할 때 자동으로 ln -s를 통하여 mm.c 파일을 링크파일로 만들어 테스트하도록 구성되어 있음

목차

1. malloc

- 1-(1) 개요(Introduction)
- 1-(2) 과제물 개요(Handout Introduction)
- 1-(3) 실습 구현하는 방법(How to Work on the Lab)
- 1-(4) Support Routines

2. mm-naive

- 2-(1) mm-naive 설명
- 2-(2) mm-naive에 사용된 함수
- 2-(3) mm-naive에 사용된 매크로

3. implicit

- 3-(1) 알고리즘
- 3-(2) implicit macro
- 3-(3) implicit source
- 3-(4) 결과 화면

1. malloc

(1) 개요(Introduction)

이번 실습에서는 C 프로그램으로 된 dynamic storage allocator을 구현하는 것이다. 즉, malloc, free, realloc, calloc 함수의 본인 버전을 구현하는 것이다.

allocator를 올바르게 구현하며, 효율적이고, 빠른 속도로 수행될 수 있도록 디자인 공간을 창의적으로 구현한다.

(2) 과제물 개요(Hand out Instructions)

malloclab-handout.tar 파일을 서버에서 자신의 directory로 복사한다.

복사한 파일을 tar xvf 명령어를 통해 압축을 풀어준다.

mm.c 파일을 수정하여 실습을 해결한다.

mdriver.c 프로그램은 구현한 프로그램의 성능을 평가하는 driver 프로그램이다. ./mdriver 명령어를 이용하여 프로그램의 성능을 평가한다.

(3) 실습을 구현하는 방법(How to Work on the Lab)

dynamic storage allocator 는 다음의 함수들을 포함한다. 이 함수들은 mm.h 에 선언되어 있고, mm.c 파일에 정의되어 있다.

```
int mm_init(void);
void *malloc(size_t size);
void free(void *ptr);
void *mm_realloc(void *ptr, size_t size);
void *mm_calloc(size_t nmemb, size_t size);
void mm_heapcheck(void);
```

주어진 mm.c 파일에는 아무것도 구현되어 있지 않다. 하지만, mm-naive.c 프로그램에는 모든 것이 정확하게 구현되어 있지만 기초적인 것들이다. 실습을 시작하기 위해 여기에 있는 코드를 mm.c 파일에 사용해도 된다. 다음의 규칙들이 만족하도록 함수들을 구현하라. (다른 private static 함수들을 정의해도 된다.)

- mm_init : 초기 힙 영역을 할당하는 것과 같은 필요한 초기화를 수행해라. 초기화를 수행하는데 문제가 있으면 1을 반환해야 한다. 다른 경우에는 0을 반환한다.

driver가 새로운 trace를 실행할 때마다, mm_init 함수를 호출함으로써 heap을 빈 heap으로 리셋시킨다.

- malloc : malloc 함수는 최소의 데이터 크기가 할당된 블록의 포인터를 반환한다. 할당된 블록 전체는 heap 영역에 있어야 하며, 할당된 다른 공간과 overlap 되어서는 안 된다.

표준 c 라이브러리 malloc 은 항상 8바이트로 정렬된 payload 포인터를 반환하므로, 마찬가지로 구현된 malloc도 항상 8바이트로 정렬된 포인터를 반환해야 한다.

- free : free 함수는 ptr이 가리키는 블록을 가용공간으로 만든다. 이 함수는 아무것도 반

환하지 않는다. malloc, calloc, realloc 함수에 의해 반환되어진 포인터이면서 free에 의해 호출되어지지 않은 인자를 넘겨받아야만 free함수는 함수 기능을 실행한다.

free(NULL)은 아무런 기능도 하지 않는다.

- realloc : realloc 함수는 다음의 제약들을 가지는 최소 바이트로 할당된 공간의 포인터를 반환한다.

- if ptr is NULL, malloc(size)와 동일하다.
- if size is equal to zero, free(ptr)과 동일하며 NULL을 반환해야 한다.
- if ptr is not NULL, 이것은 malloc 또는 realloc에 의해 반환되어진 값이며, 아직 free함수가 호출된 경우가 아니다. realloc 함수의 호출은 ptr에 의해 가리켜지는 메모리 블록의 사이즈를 size 바이트로 바꾸고, 새로운 블록의 주소를 반환한다. 구현하는 방법, old block의 내부 단편화의 양, 요구되는 크기에 따라, 새로운 블록의 주소공간은 old block 과 같을 수도 있고, 다를 수도 있다. new block의 내용은 old size와 new size 중 작은 크기 까지는 old block의 내용과 같다. 다른 것들은 초기화 되어 있지 않다.

- calloc : size 바이트를 각각 가지는 nmemb 원소 배열을 메모리에 할당한다. 할당된 메모리의 포인터를 반환한다. 이 메모리는 반환하기 전에 0으로 세트되어 있다.

- mm_checkheap : mm_checkheap 함수는 heap을 검사하고, 일관성이 있는지 검사한다. 이 함수는 malloc을 디버깅하는데 매우 유용할 것이다. 몇몇 malloc bug는 관습적인 gdb 기술을 사용하여 debug 하는데 어려움이 있다. 이러한 bug들을 해결하기 위한 효과적인 방법은 heap consistency checker를 사용하는 것이다. bug를 만났을 때, heap을 파괴하는 명령어를 찾을 때까지, consistency checker의 반복된 호출은 이 버그를 격리시킬 수 있다.

(4) Support Routines

memlib.c 에 있는 다음의 함수들을 적용할 수 있다.

- void *mem_void(int incr) : incr 바이트만큼 힙을 확장한다. incr 는 0이 아닌 양의 정수이고, 이 함수는 새롭게 할당된 heap 영역의 첫 번째 바이트에 대한 generic 포인터를 반환한다. mem_sbrk의 인자는 양의 정수인 것만 제외하면 Unix sbrk 함수와 동일하다.

- void *mem_heap_lo(void) : 힙에서 첫 번째 바이트에 대한 generic pointer를 반환.

- void *mem_heap_hi(void) : 힙에서 마지막 바이트에 대한 generic pointer를 반환.

- size_t mem_heapsize(void) : 바이트로 된 힙의 현재 크기를 반환.

- size_t mem_pagesize(void) : 바이트로 표현되는 시스템 페이지의 크기를 반환.
(4K on Linux systems)

2. mm-naive

2-(1) mm-naive 설명

- naive에 사용된 모든 함수 명을 쓰고 함수에 관련된 설명첨부
- 소스코드는 첨부하지 않아도 됨
- macro 설명 포함

2-(2) naive에 사용된 함수

int mm_init(void)

새로운 단계가 시작되면 호출되는 함수이다.

void *malloc(size_t size)

넘겨받는 인자의 크기 size 만큼 메모리를 할당해주는 함수이다.

먼저, 넘겨받은 인자 size를 정렬시킨다. 이 값을 이용하여 mem_sbrk 함수를 호출하고, heap의 크기를 늘려준다. mem_sbrk에 의해 반환되는 포인터에 SIZE_T_SIZE 만큼 증가시킨다. 크기 정보를 저장하는 부분을 SIZE_PTR(p)를 이용하여 구하고, 그곳에 크기(size)를 저장한다.

void free(void *ptr)

naive에서는 free를 사용하지 않으므로, 함수가 구현되어 있지 않다.

void *realloc(void *oldptr, size_t size)

새로운 블록을 할당해서 블록의 크기를 바꾸는 함수이다. 이전의 데이터를 복사하고, 이전의 블록을 free시킨다.

size가 0인 경우, 단지 free를 해주는 기능을 하고, NULL을 반환한다.

oldptr이 NULL인 경우, malloc을 해주는 기능을 한다.

위의 두 가지 경우가 아니고, malloc에 의해 반환되는 포인터가 null이 아니면, memcpy를 이용하여 이전 블록의 데이터를 복사하고, 이전 블록을 free 시킨다.

void *calloc(size_t nmemb, size_t size)

블록을 할당하고, 그것들을 0으로 세트시키는 함수이다. size 크기의 nmemb 만큼 메모리를 할당하고, 모두 0으로 만든다.

void mm_checkheap(int verbose)

heap을 check할 필요가 없으므로 구현되어 있지 않다.

2-(3) naive에 사용된 macro

```
#define ALIGNMENT 8
```

ALIGNMENT 상수를 정의하고 있다. 더블 워드 크기(8bytes)를 정의하고 있다.

이는 메모리 구조를 효율적으로 관리하기 위해서, 메모리 할당을 8의 배수로 할 수 있도록 한다.

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

ALIGN(size)는 주어진 size의 수에서 가장 가까운 ALIGNMENT 배수 값으로 반올림 해준다. 즉, size의 수를 ALIGNMENT 배수로 바꿔 준다. 하위 3bit와 일치하지 않는 부분만을 반환해서 ALIGNMENT 크기를 맞춘다.

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

주어진 size_t의 크기만큼 정렬(ALIGN)하고, 이 값을 SIZE_T_SIZE로 정의한다.

```
#define SIZE_PTR(p) ( ( size_t* ) ( ( char* ) (p) ) - SIZE_T_SIZE )
```

주소 p가 가리키는 블록에서 크기(size) 정보가 저장되어 있는 부분의 주소를 반환한다.

```
#define DEBUG
```

```
#ifdef DEBUG
```

```
#define dbg_printf(...) printf(__VA_ARGS__)
```

```
#else
```

```
#define dbg_printf(...)
```

```
#endif
```

DEBUG가 정의되어 있으면, 위의 dbg_printf() printf(__VA_ARGS__)를 실행하고, 아니면 dbg_printf()를 실행한다.

```
#ifdef DRIVER
```

```
#define malloc mm_malloc
```

```
#define free mm_free
```

```
#define realloc mm_realloc
```

```
#define calloc mm_calloc
```

```
#endif
```

DRIVER가 정의되어 있다면, 위의 코드들을 실행한다. 위의 코드들을 driver 테스트를 위해 각각 해당하는 파일의 가명을 정의한다.

2-(4) naive 분석

malloc을 위한 free가 구현되어야 하지만, 구현되어 있지 않다. 따라서 throughput은 높게 나오나, utilization은 나쁘게 나온다.

3. implicit 구현

3-(1) 알고리즘

implicit list는 header에 있는 size와 allocation 정보를 이용한다.

header에 있는 size를 포인터에 더해서 다음 블록으로 나아갈 수 있고, header에 있는 하위 3비트를 allocation 비트로 이용해서 블록이 할당되었는지 아닌지를 확인할 수 있다.

free block을 탐색하는 방법은 first fit, next fit, best fit 방식이 있다.

3-(2) implicit macro

<mm-implicit.c macro>

```
a201102411@eslab:~/malloclab-handout
/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(p) (((size_t)(p) + (ALIGNMENT-1)) & ~0x7)

/* masks and bitwise operators */
#define SIZEMASK (~0x7)
#define PACK(size, alloc) ((size)|(alloc))
#define getSize(x) ((x)->size & SIZEMASK)

/* implicit list macro */
/* Basic constants and macros */
#define WSIZE 4 /* word 크기를 정하고 있다 */
#define DSIZE 8 /* double word size를 정하고 있다 */
#define CHUNKSIZE (1<<12) /* 초기 heap size를 설정해준다. */

#define OVERHEAD 8 /* header + footer 사이즈. 실제 데이터가 저장되는 공간이 아니므로 overhead가 된다. */

#define MAX(x, y) ((x)>(y)?(x):(y)) /* x와 y값 중 더 큰 값 구하기*/

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size)|(alloc)) /* size와 alloc(a)의 값을 한 word로 묶는다. 이를 이용
하여 header와 footer에 쉽게 저장할 수 있다. */

/* Read and write a word at address p */
#define GET(p) (*(size_t*)(p)) /* 포인터 p가 가리키는 곳의 한 word의 값을 읽어 온다. */
#define PUT(p, val) (*(size_t*)(p)=(val)) /* 포인터 p가 가리키는 곳의 한 word의 값에 val을 기록>
한다.*/

/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p)&~0x7) /* 포인터 p가 가리키는 곳에서 한 word를 읽은 다음 하위 3bit를
버린다. 즉 header에서 block size를 읽는다. */
#define GET_ALLOC(p) (GET(p)&0x1) /* 포인터 p가 가리키는 곳에서 한 word를 읽은 다음 최하위 1bit>
를 읽는다. 0이면 블록이 할당되어 있지 않고, 1이면 할당되어 있다는 의미 */

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp) ((char*)(bp)-WSIZE) /* 주어진 포인터 bp의 header의 주소를 계산한다. */
#define FTRP(bp) ((char*)(bp)+GET_SIZE(HDRP(bp))-DSIZE) /* 주어진 포인터 bp의 footer의 주소를 계
산한다. */

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char*)(bp)+GET_SIZE(((char*)(bp)-WSIZE))) /* 주어진 포인터 bp를 이용해서
다음 block의 주소를 얻어온다. */
#define PREV_BLKP(bp) ((char*)(bp)-GET_SIZE(((char*)(bp)-DSIZE))) /* 주어진 포인터 bp를 이용해서
이전의 block의 주소를 얻어 온다. */

/* bitfields */
struct {
    unsigned allocated:1;
    unsigned size:31;
} Header;
```

83,0-1

11%

3-(3) implicit source

- int mm_init(void)

이 함수는 할당기가 초기화를 완료하고, 어플리케이션으로부터 할당과 반환 요청을 받을 준비를 완료할 수 있도록 한다.

먼저 sbrk 함수를 이용하여 4워드(16바이트)를 할당하여 빈 가용리스트를 만들 수 있도록 초기화한다.

첫 번째 워드에는 더블 워드 경계로 정렬된 미사용 패딩워드를 넣어준다.

그 다음에는 프롤로그(prologue) 블록이 온다. 이것은 header와 footer로만 구성된 8바이트 할당 블록이다. 프롤로그 블록은 초기화 과정에서 생성되며 절대 반환하지 않는다.

힙은 에필로그(epilogue) 블록으로 끝나며, 이것은 헤더로만 구성된 크기가 0으로 할당된 블록이다.

마지막 부분의 코드에서 힙을 CHUNKSIZE 바이트로 확장하고 초기 가용 블록을 생성한다.

Next fit을 이용할 경우, next_ptr을 heap_listp로 초기화 한다.

<int mm_init(void)>



```
/*
 * Initialize: return -1 on error, 0 on success.
 * 할당기는 초기화를 완료하고, 어플리케이션으로부터 할당과 반환 요청을 받을 준비를 완료한다.
 */
int mm_init(void) {
    /* Create the initial empty heap */
    /* 메모리 시스템에서 4워드를 가져와서 빈 가용리스트를 만들 수 있도록 초기화. */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void*)-1)
        return -1;
    PUT(heap_listp, 0);
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* prologue header */
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* prologue footer */
    PUT(heap_listp + (3*WSIZE), PACK(0, 1)); /* epilogue header */
    heap_listp += (2*WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    /* 힙을 CHUNKSIZE 바이트로 확장하고, 초기 가용블록을 생성 */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

- void *malloc(size_t size)

어플리케이션은 malloc 함수를 호출해서 size 바이트의 메모리 블록을 요청한다.

추가적인 요청들을 체크한 후에, 할당기는 요청한 블록 크기를 조절해서 header와 footer를 위한 공간을 확보하고, double word 요건을 만족시킨다.

최소 16바이트 크기의 블록으로 구성할 수 있도록 한다.(8바이트는 정렬 요건을 만족시키기 위해, 추가적인 8바이트는 header와 footer overhead를 위해)

8바이트를 넘는 요청에 대해서는, overhead 바이트 내에 더해주고, 인접 8의 배수로 반올림하도록 한다.

할당기가 요청한 크기를 조정한 후에, find_fit함수를 이용하여 적절한 가용 블록을 가용 리스트에서 검색한다. 만일 맞는 블록을 찾으면 할당기는 place 함수를 이용하여 요청한 블록을 배치하고, 옵션으로 초과부분을 분할하고, 새롭게 할당한 블록을 반환한다.

만일 할당기가 맞는 블록을 찾지 못하면(메모리가 부족하다면), extend_heap을 이용하여 힙을 새로운 가용 블록으로 확장하고(메모리 공간을 더 할당), 요청한 블록을 이 새 가용 블록에 배치하고, 필요한 경우에 블록을 분할하고, 이후에 새롭게 할당한 블록의 포인터를 리턴한다.

<void *malloc(size_t size)>

```
a201102411@eslab:~/malloclab-handout
/*
 * malloc : size 바이트의 메모리 블록을 요청
 */
void *malloc (size_t size) {
    size_t asize; /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size==0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    /* 최소 16 바이트 크기의 블록 구성
     * 8 바이트는 정렬 조건 만족시키기 위해
     * 추가적인 8 바이트는 헤더와 풋터 오버헤드를 위해
     */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else // 8바이트 넘는 요청 : 오버헤드 바이트 내에 더해주고 인접 8의 배수로 반올림.
        asize = DSIZE*((size+(DSIZE)+(DSIZE-1))/DSIZE);

    /* Search the free list for a fit*/
    /* 적절한 가용블록을 가용리스트에서 검색 */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        // 맞는 블록 찾으면 할당기는 요청한 블록 배치하고, 옵션으로 초과부분을 분할.
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    // 힙을 새로운 가용 블록으로 확장
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

- static void *coalesce(void *bp)

인접 가용 블록들을 통합시키는 함수이다.

할당기가 현재 블록을 반환할 때 가능한 모든 경우는 다음과 같다.

- (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.

첫 번째 경우는 인접 블록 모두가 할당 상태이므로 연결이 불가능하다.

두 번째 경우는 현재 블록이 다음 블록과 통합된다. 현재 블록과 다음 블록의 footer는 현재와 다음 블록의 크기를 합한 것으로 갱신된다.

세 번째 경우는 이전 블록이 현재 블록과 통합된다. 이전 블록의 header와 현재 블록의 footer는 두 블록의 크기를 합한 것으로 갱신된다.

네 번째 경우는 세 블록 모두는 하나의 가용 블록으로 통합된다. 이전 블록의 header와 다음 블록의 footer는 세 블록의 크기를 합한 것으로 갱신된다.

```
// next fit
if ((next_ptr > (char*)bp) && (next_ptr < NEXT_BLKPTR(bp)))
    next_ptr = bp;
```

Next fit을 이용할 경우, 위의 코드를 추가한다.

<static void *coalesce(void *bp)>

```
a201102411@eslab:~/malloclab-handout
static void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKPTR(bp)));
    // 이전 블록의 할당 여부 0 = no, 1 = yes
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(bp)));
    // 다음 블록의 할당 여부 0 = no, 1 = yes
    size_t size = GET_SIZE(HDRP(bp));
    // 현재 블록의 size
    /*
     * case 1 : 이전 블록, 다음 블록 최하위 bit 둘 다 1 할당
     * 블록 병합 없이 bp return
     */
    if (prev_alloc && next_alloc) {
        return bp;
    }
    /*
     * case 2 : 이전 블록 최하위 bit 1(할당), 다음 블록 최하위 bit 0(비할당)
     * 다음 블록과 병합한 뒤 bp return
     */
    else if (prev_alloc && !next_alloc) {
        size += GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    /*
     * case 3 : 이전 블록 최하위 bit 0(비할당), 다음 블록 최하위 bit 1(할당)
     * 이전 블록과 병합한 뒤 새로운 bp return
     */
    else if (!prev_alloc && next_alloc) {
        size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
        bp = PREV_BLKPTR(bp);
    }
    /*
     * case 4 : 이전 블록 최하위 bit 0(비할당), 다음 블록 최하위 bit 0(비할당)
     * 이전 블록, 현재 블록, 다음 블록 모두 병합한 뒤 새로운 bp return
     */
    else {
        size += GET_SIZE(HDRP(PREV_BLKPTR(bp))) + GET_SIZE(FTRP(NEXT_BLKPTR(bp)));
        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
        bp = PREV_BLKPTR(bp);
    }
    return bp;
}
```

- void free(void *ptr)

이전에 할당한 블록을 free 함수를 호출해서 반환한다. 이것은 요청한 블록을 반환하고, 인접 가용 블록들을 경계 태그 연결 기술을 사용해서 통합한다.

<void free(void *ptr)>

```
a201102411@eslab:~/malloclab-handout
/*
 * free : 이전에 할당한 블록을 반환한다.
 */
void free(void *ptr) {
    /* 잘못된 free 요청일 경우 함수 종료, 이전 프로시저로 return */
    if(!ptr) return;

    size_t size = GET_SIZE(HDRP(ptr)); // ptr의 헤더에서 block size 읽어 옴

    /* 실제로 데이터를 지우는 것이 아니라
     * header와 footer의 최하위 1비트(할당된 상태)만을 수정 */
    PUT(HDRP(ptr), PACK(size, 0));
    // ptr의 header에 block size와 alloc = 0을 저장
    PUT(FTRP(ptr), PACK(size, 0));
    // ptr의 footer에 block size와 alloc = 0을 저장
    coalesce(ptr);
    // 주위에 빈 블록이 있을 시 병합.
}
```

-static void *extend_heap(size_t words)

이 함수는 새 가용 블록으로 힙을 확장하는 함수이다.

extend_heap 함수는 힙이 초기화될 때, malloc이 적당한 맞춤(fit)을 찾지 못했을 때 정렬을 유지하기 위해서 호출된다.

먼저, 요청한 크기를 인접 2워드의 배수로 반올림하며, 그 후에 sbrk를 이용하여 메모리 시스템으로부터 추가적인 힙 공간을 요청한다.

그리고 가용 블록의 header와 footer, epililogue header를 초기화한다.

마지막 부분은 이전 힙이 가용 블록으로 끝났다면, 두 개의 가용 블록을 통합하기 위해 coalesce 함수를 호출하고, 통합된 블록의 포인터를 반환한다.

<static void *extend_heap(size_t words)>



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

    /* Allocate an even number of words to maintain alignment */
    /* 요청한 크기를 인접 2워드의 배수로 반올림하며,
     * 그 후에 메모리 시스템으로부터 추가적인 힙 공간 요청
     */
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0)); /* Free block header */
    PUT(FTRP(bp), PACK(size, 0)); /* Free block footer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}
```

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

이 함수는 요청한 블록을 가용 블록의 시작 부분에 배치하며, 나머지 부분의 크기가 최소 블록 크기와 같거나 큰 경우에만 분할한다.

블록을 배치한 후에 나머지 부분의 크기가 최소 블록 크기와 같거나 크다면, 블록을 분할해야 한다. 다음 블록으로 이동하기 전에 새롭게 할당한 블록을 배치해야 한다.

<static void place(void *bp, size_t asize)>

```
a201102411@eslab:~/malloclab-handout
static void place(void *bp, size_t asize) {
    size_t csize = GET_SIZE(HDRP(bp));

    /*
     * 배치 후에 이 블록의 나머지가 최소 블록 크기와 같거나 크다면,
     * 진행해서 블록을 분할해야한다.
     */
    if ((csize - asize) >= (2*DSIZE)) {
        /* 새롭게 할당된 블록을 배치 */
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLK(bp); /* 다음 블록으로 이동 */
        PUT(HDRP(bp), PACK(csize - asize, 0));
        PUT(FTRP(bp), PACK(csize - asize, 0));
    }
    else { /* 같은 경우엔 전체를 할당한다 */
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

- static void *find_fit(size_t asize) : First fit 사용

First fit을 이용하여 fit을 찾는 함수이다.

First fit은 리스트의 첫 부분부터 검색하면서 해당하는 size의 블록을 찾는 방법이다.

이는 반복문을 이용하여 구현할 수 있다.

<static void *find_fit(size_t asize)>

```
a201102411@eslab:~/malloclab-handout
static void *find_fit(size_t asize) {
    /* First fit search */
    void *bp;

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLK(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    return NULL; /* No fit */
}
```

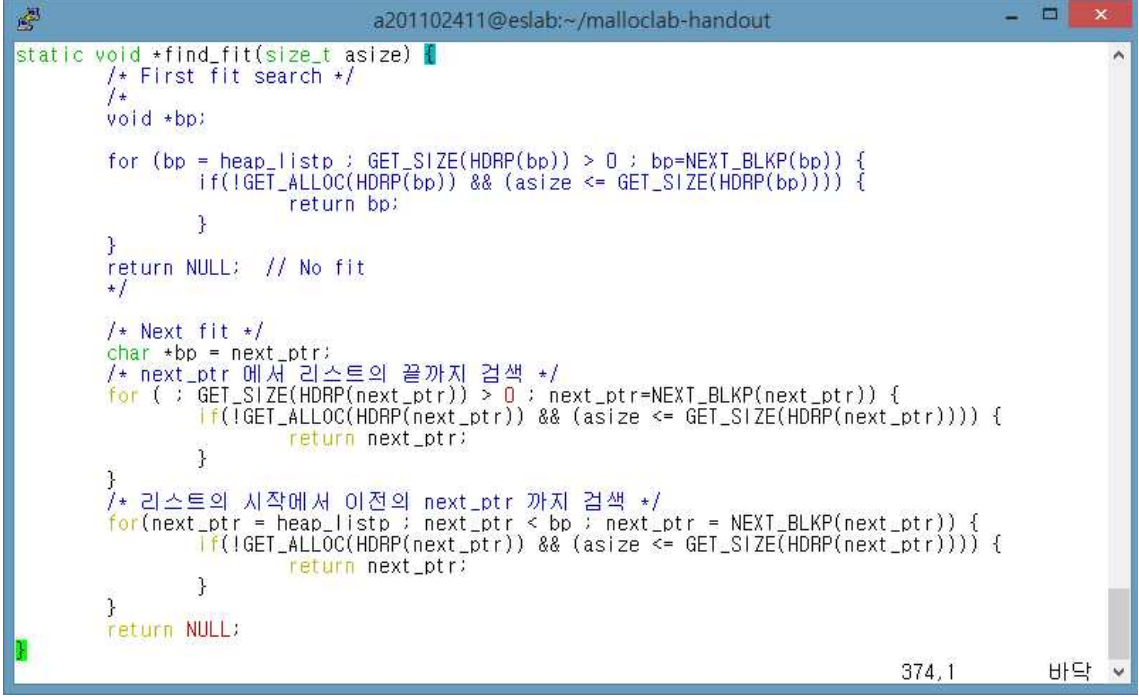
- static void *find_fit(size_t asize) : Next fit 사용

Next fit을 이용하여 fit을 찾는 함수이다. Next fit은 이전 검색이 종료된 지점에서 검색을 시작한다. 이를 위해 변수 next_ptr을 지정한다.

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.

<static void *find_fit(size_t asize)>



```
static void *find_fit(size_t asize) {
    /* First fit search */
    /*
    void *bp;

    for (bp = heap_listp ; GET_SIZE(HDRP(bp)) > 0 ; bp=NEXT_BLKp(bp)) {
        if(!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    return NULL; // No fit
    */

    /* Next fit */
    char *bp = next_ptr;
    /* next_ptr 에서 리스트의 끝까지 검색 */
    for ( ; GET_SIZE(HDRP(next_ptr)) > 0 ; next_ptr=NEXT_BLKp(next_ptr)) {
        if(!GET_ALLOC(HDRP(next_ptr)) && (asize <= GET_SIZE(HDRP(next_ptr)))) {
            return next_ptr;
        }
    }
    /* 리스트의 시작에서 이전의 next_ptr 까지 검색 */
    for(next_ptr = heap_listp ; next_ptr < bp ; next_ptr = NEXT_BLKp(next_ptr)) {
        if(!GET_ALLOC(HDRP(next_ptr)) && (asize <= GET_SIZE(HDRP(next_ptr)))) {
            return next_ptr;
        }
    }
    return NULL;
}
```

374,1 바닥

3-(4) 결과 화면

mdriver를 통해 성능을 비교해보면, naive 보다는 first fit의 성능이 좋다는 것을 알 수 있다. 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.

implicit 리스트 구조의 특성상 모든 블록을 탐색하는 best fit 방법은 throughput이 좋지 않다.

<First fit 결과>

```
[a201102411@eslab malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3392.3 MHz

Results for mm malloc:
  valid  util   ops    secs    Kops  trace
  yes    34%    10  0.000000  55887 ./traces/malloc.rep
  yes    28%    17  0.000000  93166 ./traces/malloc-free.rep
  yes    96%    15  0.000000  45760 ./traces/corners.rep
* yes    86%   1494  0.001210   1235 ./traces/perl.rep
* yes    75%    118  0.000012   9965 ./traces/hostname.rep
* yes    91%  11913  0.071505    167 ./traces/xterm.rep
* yes    99%   5694  0.007262    784 ./traces/amptjp-bal.rep
* yes    99%   5848  0.006712    871 ./traces/cccp-bal.rep
* yes    99%   6648  0.013282    501 ./traces/cp-decl-bal.rep
* yes   100%   5380  0.010649    505 ./traces/expr-bal.rep
* yes    66%  14400  0.000138104261 ./traces/coalescing-bal.rep
* yes    93%   4800  0.011268    426 ./traces/random-bal.rep
* yes    55%   6000  0.028177    213 ./traces/binary-bal.rep
10      86%  62295  0.150214    415

Perf index = 56 (util) + 17 (thru) = 72/100
[a201102411@eslab malloclab-handout]$
```

<Next Fit 결과>

```
a201102411@eslab:~/malloclab-handout
[a201102411@eslab malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 3392.3 MHz

Results for mm malloc:
  valid  util   ops    secs    Kops  trace
  yes    34%    10  0.000000  77985 ./traces/malloc.rep
  yes    28%    17  0.000000  87911 ./traces/malloc-free.rep
  yes    96%    15  0.000000  61904 ./traces/corners.rep
* yes    81%   1494  0.000048  31319 ./traces/perl.rep
* yes    75%    118  0.000001100049 ./traces/hostname.rep
* yes    91%  11913  0.000758  15715 ./traces/xterm.rep
* yes    91%   5694  0.001696   3357 ./traces/amptjp-bal.rep
* yes    92%   5848  0.001066   5488 ./traces/cccp-bal.rep
* yes    95%   6648  0.004866   1366 ./traces/cp-decl-bal.rep
* yes    97%   5380  0.005338   1008 ./traces/expr-bal.rep
* yes    66%  14400  0.000140103054 ./traces/coalescing-bal.rep
* yes    91%   4800  0.007232    664 ./traces/random-bal.rep
* yes    55%   6000  0.002720   2206 ./traces/binary-bal.rep
10      83%  62295  0.023864   2610

Perf index = 54 (util) + 40 (thru) = 94/100
[a201102411@eslab malloclab-handout]$ vi mm-implicit.c
[a201102411@eslab malloclab-handout]$
[a201102411@eslab malloclab-handout]$
[a201102411@eslab malloclab-handout]$
[a201102411@eslab malloclab-handout]$
```